JADE: COMPILER-SUPPORTED MULTI-PARADIGM PROCESSOR VIRTUALIZATION-BASED PARALLEL PROGRAMMING

BY

JAYANT DESOUZA

B.E., University of Madras, 1993
M.S., Kansas State University, 1997

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2004

Urbana, Illinois

Abstract

Current parallel programming approaches, which typically use message-passing and shared-memory threads, require the programmer to write considerable low-level work management and distribution code to partition and distribute data, perform load distribution and balancing, pack and unpack data into messages, and so on. One solution to this low level of programming is to use *processor virtualization*, wherein the programmer assumes a large number of available virtual processors and creates a large number of work objects, combined with an adaptive runtime system (ARTS) that intelligently maps work to processors and performs dynamic load balancing to optimize performance. Charm++ and AMPI are implementations of this approach. Although Charm++ and AMPI enable the use of an ARTS, the program specification is still low-level, requiring many details. Furthermore, the only mechanisms for information exchange are asynchronous method invocation and message passing, although some applications are more easily expressed in a shared memory paradigm.

We explore the thesis that compiler support and optimizations, and a disciplined shared memory abstraction can substantially improve programmer productivity while retaining most of the performance benefits of processor virtualization and the ARTS.

The ideas proposed in this thesis are embodied in a new programming language, Jade, based on Java, Charm++ and AMPI. The language design uses the Java memory model, automating memory management and eliminating void pointers and pointer arithmetic. In addition, by automating various routine tasks in Charm++, programmer productivity is substantially improved. Jade introduces Multiphase Shared Arrays (MSA), which can be shared in `read-only`, `write-many`, and `accumulate` modes. These simple modes scale well

and are general enough to capture the majority of shared memory access patterns.

We present novel uses of known compilation techniques, as well as new compile-time analyses suggested by the needs of ARTS. One optimization strip-mines MSA loops and optimizes away a test that checks if a page is present in the local MSA cache, resulting in single-cpu MSA performance matching that of a sequential program. Another optimization generates guarded pack/unpack code that only packs live data. This significantly reduces the time taken and disk size needed to checkpoint (or migrate objects within) large applications.

The Jade language and compiler system described in this thesis can serve as the framework for further research into compiler-based multi-paradigm ARTS-supported parallel programming built upon processor virtualization.

In memory of my father,

Arthur DeSouza, 1923–1978.

I wish you were here.

Acknowledgments

God.

My dream of completing a PhD would not have come to fruition without Prof. Kalé's patient support. It has been a long journey requiring tremendous sacrifice and personal struggle; and now when I look back, I realize how much the journey has changed me, and how much Prof. Kalé has contributed to the process.

My undergraduate teachers, Prof. N. Venkateswaran, and Dr. B. Ramesh represented quality and scholarship and were the role models for my aspirations.

The guidance of my thesis committee has been invaluable in focusing the research described in this dissertation. I warmly thank Profs. Vikram Adve, David Padua, and Marc Snir for their assistance.

The group members of the Parallel Programming Laboratory are a continual source of exciting discussions and tremendous expertise on parallel computing and the Charm++ system. Those I have had much interaction with include: Sanjeev Krishnan, Milind Bhandarkar, Robert Brunner, Orion Sky Lawlor, Sameer Kumar, Gengbin Zheng, Terry Wilmarth, Chee Wai Lee, Mark Hills, Eric Bohm, Sayantan Chakravorty, Rahul Joshi, and the Faucets team: Justin Meyer, Sindhura Bandhakavi, Mani Potnuru. I thank them all for their contributions,

direct and indirect. [1]

A special thanks to Prof. Klaus Schulten, and the members of the Theoretical and Computational Biophysics Group, especially Kirby Vandivort, Jim Phillips, and the ultimate sysadmin, Tim Skirvin.

I wish to also thank Jay Hoeflinger at Intel, Champaign for putting me on the TreadMarks project during my internship there. The experience I acquired with DSM systems (including one assembly-level bug hunt) helped me debug the complicated parallel behavior in the implementation of MSA.

My wife, Sonali, believed in me when I doubted myself. She stood by me when one year became two, and then three, and then four, to complete the PhD. Words cannot describe how grateful I am to have her in my life. She is my rock.

My daughter, Aisha, came along in 2001. This little bundle of joy became my reason why.

My family: my mother, Wynnie, brothers Anand and Mohan, and sister Maya.

And last, but certainly not least, my in-laws, Anthony and Nona D'Almeida.

[1]This work was supported in part by the National Institute of Health under Grant NIH PHS 5 P41 RR05969-04 and the local Department of Energy ASCI center under Grant DOE B341494.

Some computations were performed on the National Science Foundation Terascale Computing System (lemieux) at the Pittsburgh Supercomputing Center (PSC).

This work was partially supported by National Computational Science Alliance (NCSA) under [grant number] and utilized the platinum and tungsten systems.

Some computations were performed on the architecture cluster at the University of Illinois, which is funded by the National Science Foundation under grant NSF EIA 02-24453.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

A promising approach to parallel programming that the Parallel Programming Laboratory (PPL) has been exploring for the past decade is to effect an "optimal" division of labor between the application programmer and the programming system. *Processor virtualization*(PV), which requires the programmer to decompose their application into a large number of interacting objects, is a foundational part of this approach. The adaptive runtime system (ARTS) is responsible for mapping the objects to processors and managing them. The "objects" may be true C++ objects, or they may be user-level threads running MPI[12]. In either case, the model requires strict encapsulation: each object is allowed to access only its own variables and only accesses other objects' data via well-defined interfaces such as remote method invocation, message passing, or certain information sharing abstractions such as `readonly` data.

This virtualization approach leads to several benefits. It provides powerful scalability, efficient execution on both shared and distributed memory architectures and a simple programmer cost model for the various features of the language. Most importantly, it allows the ARTS to do object migration and automatic dynamic measurement-based load balancing [33, 17], automatic checkpointing, out-of-core execution, dynamically shrink or expand the set of processors used by the job[38], etc.

This approach has been implemented in C++, in the form of the Charm++ system[39, 34, 36]. Charm++ has been very successful in parallelizing several applications effectively.

As an example, NAMD[37], a molecular dynamics application written in Charm++ is highly scalable and runs on thousands of processors[57]. Recently, NAMD won the Gordon Bell award at SC2002[58]. Another example is the FEM framework[11] written in Charm++ and AMPI, which is being used to develop very large applications at the Center for the Simulation of Advanced Rockets.

We explore the thesis that programming productivity in processor virtualization languages can be significantly enhanced by extending the Charm++/AMPI model to allow a restricted form of shared-address-space programming, and by using compiler supported source-to-source translation. We design a new programming language, Jade, that overcomes several limitations of current PV languages including Charm++ and AMPI. We add a disciplined yet sufficiently general shared-address space (SAS) programming model called multiphase shared arrays (MSA) to Jade. This rounds out the multi-paradigm capabilities of virtualization technology. Jade is implemented using a full multi-pass compiler framework. Although difficult to quantify, we believe that programming in Jade is considerably easier and less prone to error than programming in current PV languages. We provide several example programs in this thesis to serve as empirical evidence of this claim. We also compare code sizes of equivalent Charm++ and Jade programs and discuss the kinds of errors eliminated in Jade and the debugging assistance that Jade provides.

As discussed below, we build on the Jade compiler framework and implement a strip-mining optimization that resolves an intrinsic performance issue with the MSA model, placing its performance on par with that of directly accessed local sequential arrays. We identify several optimizations that are made possible by the Jade compiler system and implement one of them: a mechanism to reduce the size and time taken to checkpoint objects by only saving data that is still live.

We discuss the above points in more detail below.

The lack of basic compiler support in current PV languages such as Charm++ and AMPI has necessitated placing several restrictions and requirements on the programmer.

Programmers must maintain a special interface translator file for each program module, in which Charm++ entities are listed. This is in addition to their source code files and they must keep it synchronized with their class and method names and signatures. No type-checking is performed; errors can result in strange behavior the cause of which is very difficult to track down. Arcane notation and conventions must be followed to create and initialize parallel objects (such as `chares`, `ChareArrays` and `readonly` variables). Tedious, repetitive code needs to be written to pack and unpack object data so that objects can migrate. The original Charm++ translator (prior to work done for this thesis) did not support parameter marshalling. `Message`'s were the unit of communication in Charm++ and the programmer needed to define a `Message` for each kind of communication, and manage the creation and destruction of `Message` objects. Inconsistent behavior is found in the Charm++ language.[1] The need for even a simple compiler has been felt repeatedly.

Taking our focus on improving programmer productivity further, we found that pointers and the C++ style memory management of Charm++ often led to complex bugs; and `void` pointers and pointer arithmetic in C++ give rise to the aliasing issue which complicates compiler analysis in general. The Java[2] memory model appears to solve these issues at the cost of the overhead of garbage collection[15].

We initially studied implementing a Charm++ compiler based on C++ (without pointers). But an issue with implementing a compiler for Charm++ is the complexity of supporting the C++ language: pointers, operator-overloading, friends, general templates, multiple inheritance, destructors, enums, namespaces, reference variables and parameters, exceptions, preprocessing directives, etc. The resources of a research group are limited and we decided that maintaining a full-fledged C++ compiler would not provide sufficient benefit to justify the needed resources.

[1] For instance, allowing 1D but not 2D ChareArrays to be declared with their size in the constructor; using `Message`'s for reduction callbacks instead of data types; etc.

[2] Java is a trademark of Sun Microsystems, Inc. All other trademarks are properties of their respective owners.

Therefore, we designed the Jade programming language. Jade is a simple language based on Java and Charm++/C++. It is easier to compile and analyze than C++ or Java. It has strongly-typed "pointers" in the form of references; the intent of this is to permit better compiler analysis in the long term. The Jade design uses garbage collection for memory management, which enhances ease of programming. Jade also provides block multi-dimensional sequential arrays (as opposed to Java's array of arrays) and supports HPF-like array section notation. Jade addresses all the basic usability issues mentioned above; these are described in more detail in Chapter 2. Jade is implemented as a multi-pass compiler framework that performs forward reference resolution, type-checking, one or more optimization passes, and a code generation pass that generates Charm++ code. The compiler framework provides a base for future enhancements as needed.

A strength of the virtualization-based Charm++ system is that it supports multiple programming paradigms, such as the object-oriented message-driven style of the Charm++ language and the message-passing style of AMPI. However, Charm++ does not support a general shared address space (SAS) model. In some cases SAS programs are hard to develop and face difficulties due to a large number of race conditions, but in other cases they are easier to develop, and we wanted to make this programming paradigm available in Jade to improve productivity in PV programming systems.

General shared address space parallel programming models have faced difficulty scaling to large numbers of processors. Distributed Shared Memory (DSM) is a much-studied[31, 1] software-level shared memory solution that runs into the same problem. Optimized cache coherence mechanisms based on the specific access pattern of a shared variable show significant performance benefits over general DSM coherence protocols[8, 9]. Charm++ supports a very limited "shared memory" style in the form of information sharing abstractions such as readonly, monotonic, and accumulator variables.[3]

[3]Monotonic and accumulator variables have fallen into disuse, and only readonly variables are being actively used.

We suggest that a disciplined form of shared-memory programming that takes specifically shared variables further can round out and "complete" virtualization-based programming without sacrificing performance. We describe Multiphase Shared Arrays (MSA), a system that supports such specifically shared arrays that can be shared in `read-only`, `write-many` (where each location is updated by at most one thread), `accumulate` (where multiple threads may update a single location, but only via a well-defined commutative associative operation), and `owner-computes` modes. These simple modes scale well and are general enough to capture the majority of shared memory access patterns. MSA does not support a general `read-write` access mode. MSA coexists with the message-passing paradigm (MPI) and the message-driven paradigm(Charm++). We present the MSA model, its implementation in Jade and as a library in Charm++, programming examples and performance results in Chapter 3.

The compiler-based Jade approach also enables other optimizations that are difficult to achieve in current PV languages. As described in Chapter 4, by keeping track of dead variables, we use the compiler to generate guarded pack/unpack code that only packs up live data. This significantly reduced the time taken and disk size needed to checkpoint (or migrate objects within) the studied Jacobi application.

Another example of how compiler support can be utilized is provided by the following MSA optimization. The default implementation of accessing MSA array elements in `read-only` mode is slow because every read access requires a check to determine whether the element is available locally. DSM systems use the virtual memory(VM) page fault mechanism to efficiently perform the same test, but a feature of MSA that we wish to retain is its variable-sized pages that are not tied to the hardware VM page size. We used the Jade compiler to strip-mine loops reading MSA arrays and optimize away the check. As a result, we were able to obtain single-cpu MSA performance matching that of a sequential program.

We believe that the Jade language and compiler framework with the MSA abstraction implements a "complete" and powerful processor virtualization-based parallel programming

system while significantly improving ease of programming of such systems and opening the door to compiler-based optimization of PV systems.

Chapter 2

Jade

Jade is a Java-like language built to improve productivity in the programming of processor virtualization-based (PV) applications. Jade incorporates design features and uses compiler and runtime support to simplify programming and automate routine tasks required of the programmer by existing PV languages. Jade rounds out and completes current-generation PV languages by supporting a specifically shared memory programming paradigm in addition to existing distributed memory PV models. Jade is implemented as a multi-pass compiler framework that includes optimization passes described in Sections 3.3 and 4.1.

Jade uses Java's syntax and basic sequential semantics, and extends them with Charm++, AMPI, and multiphase shared array (MSA) parallel constructs. Multiphase shared arrays (MSA), described in Chapter 3, provide a virtualization-based shared address space (SAS) programming model that complements the distributed models of Charm++ and AMPI. Jade parallel objects can be of various kinds: message-driven Charm++ objects, message-driven or message-passing user-level threads (including AMPI threads), or SAS MSA array objects. This multi-paradigm programmability in Jade allows the programmer to express each component of their application in the most natural programming paradigm, leading to improved productivity.

Jade borrows its memory model from Java and does away with pointer arithmetic. Java's memory model and garbage collection make memory management easier. This design decision removes a major source of difficult bugs, thereby further improving programmer produc-

tivity. Boehm[15] discusses in detail the advantages of using garbage collection, and refers to studies that show that 30%-40% of development time for complex linked data structures is devoted to storage management. The absence of void pointers and pointer arithmetic in Jade enables better compiler analysis because the aliasing issue is removed.

Jade is closely based on Charm++ for its parallel entities and semantics while using a compiler approach to improve the syntax significantly and automate routine tasks. The key syntactic productivity-enhancing features implemented in Jade are:

- *Parameter marshalling*: Charm++ did not support parameter marshalling[1], but used a special entity called a `Message` as the unit of communication and information exchange. The user needs to define a `Message` for each kind of communication, and manage the creation and destruction of `Message` objects. Jade implements a proxy-wrapper mechanism for each parallel object, and automatically generates code to pack parameters into a `Message` and unpack it on the receiving side. In addition to the primitive types, parameters can be arrays or complex objects containing dynamically-allocated data, as well as references/handles/proxies to parallel objects. Parameter passing in described in detail in Section 2.2.1.

- *Automated Migration*: To support migration and load-balancing in Charm++, the user needs to write pack-unpack methods for all classes that might be migrated. This step has been considerably simplified with the `pup` framework in Charm++ now[32]. However, some users find writing the `pup` methods for each class cumbersome. Without this step, the advantages of automatic load-balancing will not be realized. Since these methods can be auto-generated from a list of the data members of a class and since Jade does not have to deal with the aliasing issues associated with pointers in Charm++, this is done automatically for the user in Jade. Therefore, parallel objects

[1]Our approach has been to implement a feature directly in Charm++ as far as possible, in order to make it available to Charm++ programmers. We then design the feature in the Jade language at a higher level, and translate Jade to Charm++. Some of the features listed may therefore also be available in Charm++, but only as a byproduct of the Jade implementation.

in Jade are automatically capable of migration and can participate in load-balancing and checkpointing with no further effort from the programmer.

Jade is implemented as a multi-pass compiler framework that performs forward reference resolution, type-checking, one or more optimization passes, and a code generation pass. Jade is translated to Charm++ code, which is then compiled to binary/object code and executed on the target machine. The resulting translated code runs on all the platforms supported by Charm++, such as Linux or Windows machines with Ethernet, Myrinet, Quadrics, or Infiniband interconnects; IBM SP machines; Compaq AlphaServer machines; and any machine that supports MPI.

We cover the parallel part of Jade in detail in Section 2.1 but defer the details of multiphase shared arrays (MSA) to Chapter 3. We then describe the major productivity-enhancing contributions of Jade in the next section, followed by the highlights of the Jade features borrowed from Java. Java's parameter passing semantics are the source of considerable confusion, and so in Section 2.2.1 we clarify the assignment and parameter passing semantics of Jade. Translator issues are mentioned in Section 2.4. In Section 2.5, we present some example programs written in Jade, and compare them to their Charm++ equivalents to show how productivity is enhanced in Jade. The performance of Jade programs is studied in Section 2.6.

2.1 Jade Programming Abstractions

Programming in a multi-paradigm parallel language like Jade requires knowledge of several concepts. We describe these concepts in this section. We cover the execution model based on virtualization and the adaptive runtime system. We then describe the communication model and message delivery. We present the parallel constructs of Jade and then discuss how various programming models can be realized in Jade.

2.1.1 Jade Virtualization-based Object Model

Since Jade is a multi-paradigm programming language, it contains several programming models. The smooth interaction of entities from various paradigms is made possible in large part by the unifying concept of processor virtualization. Since Jade is object-oriented, we often use the terms "object" or "work object" to refer to parallel entities.

For the most part, the Jade programmer is insulated from the system hardware. The programmer's view, as in Charm++, is that of a collection of parallel objects executing on a set of processors. In the Jade programming model, the programmer does not know or care which object is on which processor or how many processors there are. Parallel objects can migrate among processors and in general, the user cannot assume the location of an object.

An adaptive runtime system (ARTS) is a core part of the Jade system model. The ARTS has a presence on each node and runs a "scheduler" on each processor of the node. The scheduler's job is to manage the messages intended for objects on the processor. The ARTS also includes a load-balancing component that measures various system parameters and application behavior and migrates parallel objects to balance the system and communication load.

Each parallel object tightly encapsulates its data: no `public` data members are permitted. Specific information sharing abstractions (Section 2.1.3) enable data sharing.

Each parallel object has a set of public methods that can be invoked on it. The methods can take parameters which are passed by value. Methods may be threaded or non-threaded (the default) and are described in more detail in Section 2.1.2. We often use the Charm++ term `entry method` to refer to the public methods of a parallel object as distinguished from the public methods of a sequential class.

The system architecture assumed in Jade is that of a set of nodes connected by a network interconnect. Each node has one or more processors and local memory which it can access. In the case of SMP nodes containing multiple processors with a single local memory, the

Jade system model treats each processor as an individual entity with its own local memory. Remote memory is not directly accessible.

2.1.2 Jade Communication Model

Methods of a parallel object can be declared as threaded or non-threaded (the default). Each threaded method is executed in its own user-level thread. Threads are not preemptible. Threaded methods can call `CthYield` to allow other threads to run. Many Jade system calls (such as `recv`) also implicitly yield.

Asynchronous Method Invocation (AMI)

Method invocations are asynchronous by default. Thus if object A invokes a method (whether threaded or not) on object B, the call is asynchronous. i.e. control returns immediately to the caller and the ARTS handles message delivery. The caller does not wait but continues execution immediately.

AMI is just like sending a message; but the associated features of the message-passing programming style popularized by MPI are not present, e.g. in Jade there is no corresponding blocking receive for messages.

Note that AMI does not directly permit traditional blocking function calls (possibly with return values). This must be achieved by splitting up the calling method into two methods, the first of which contains the code up to and including the AMI call. The called function must then send back a message to the second method to resume the computation, as shown in the example below. How does the called function know who to call back? One of its incoming parameters can contain the handle of the caller. Another option is to pass in a `CkCallback` object (as described in the Charm++ manual).

```
1 chare A {
2   public void caller1() {
3     // do work
4     // now we need something from B
```

```
5       B b = new B();
6       b.callee(this);  // send B a handle to myself
7     }
8
9     public void caller2(int response) {
10      // continue
11    }
12  }
13
14  chare B {
15    private int reply;
16    public void callee(A a) {
17      a.caller2(reply);
18    }
19  }
```

Synchronous Method Invocation

Synchronous method invocation is specified at the method level; it is controlled by adding the `sync` keyword to the called method. Invoking a `sync` method is synchronous, i.e. the calling function blocks until the called method completes execution and returns. Therefore, only threaded methods can call `sync` methods.

This behavior requires the programmer to look at the signature of the called function to determine the behavior at the calling site. This restriction could be removed by having the Jade compiler generate a `sync` and a non-`sync` version of each method. The calling site can then indicate whether a `sync` call or a non-`sync` call is intended and the compiler can invoke the appropriate method.

```
1  // caller
2  sync B.callee();  // blocks
3  B.callee(); // does not block
```

Message Delivery and Concurrency Within Objects

A parallel object lives (executes) on only one processor. When an object A invokes a method on a parallel object B, the ARTS determines the location of B and ensures the message is delivered to it. Messages are not guaranteed to be delivered in any particular order. Messages

are enqueued in the message queue of the processor on which an object resides. The scheduler picks the first message off the queue and invokes the appropriate method on the target object. Execution continues until the method yields or terminates.

An implication of this is that only one of the non-threaded methods of an object executes at any given time. Therefore, if a class contains only non-threaded methods, no locking is required when accessing the class data. Another implication of this is that a threaded method of an object can interleave execution with other threaded or non-threaded methods of the same object, but two methods of an object cannot execute simultaneously (on different processors). Locking may be required if one threaded method and another method modify the same class data.

2.1.3 Jade Parallel Constructs

Jade/Charm++ parallel constructs include parallel classes called *Chares* and parallel arrays of objects called *ChareArrays*. In their basic form, these entities are message driven, which means they do not maintain their own thread of control. A *scheduler* that runs on each processor grants an object control by invoking one of its methods; the object executes the method and returns control to the scheduler. In the message-driven paradigm, when a method of a `chare` or `ChareArray` is invoked, it continues to completion before any other method of the same parallel object is permitted to run.

Another Charm++ construct that Jade supports is user-level threads – an object's methods may be marked as `threaded`, which means they execute in a user-level thread that can block.

Finally, Jade supports specifically shared MSA arrays, which provide a restricted form of shared-memory programming.

```
 1 /** Simple parallel hello program using ChareArrays.
 2  main sends a message to hi[0], which sends it on to the
 3    next element, and so on.*/
 4 package Hello;
 5
 6 public synchronized class TheMain extends Chare {
 7     public int main(String []args){
 8         ChareArray hi1 = new Hello[5];
 9         hi1[0].sayHello(21);
10     }
11
12     public void memberMethod(int parameter) {...}
13 }
14
15 public synchronized class Hello extends ChareArray1D {
16     public Hello(){}
17
18     public void sayHello(int number) {
19         if (thisIndex < 4)
20             thisProxy[thisIndex+1].sayHello(number+1);
21         else
22             CkExit();
23     }
24 }
```

Figure 2.1: Example Jade program: `ChareArray` Hello.

Program Startup and Termination

Figure 2.1 shows an example Jade program. Line 4 declares the Hello package which contains the subsequent entities. If no package is declared, the entities belong to a default null package.

Jade execution model: Execution of a Jade program begins by instantiating one copy of every chare that has a `main` (see line 7 of Figure 2.1), and then invoking the `main` function of the chare on processor 0. This is different from the standard Java behavior, in which `main` is `static` and only one `main` is invoked, without instantiating the class containing it. Typically, the `main` functions create parallel objects and invoke methods on them to get the work started. The reason we instantiate the class containing `main` is because most programs need an object to send back results to, print them out, etc. Usually one `main` is sufficient for a program, but multiple `main`'s are supported for Jade module initialization. Note that `ChareArrays` (described below) cannot have `main` functions.

Unlike traditional sequential languages, (but similar to threaded languages) `main` can

terminate without causing the program to end. Thus, on line 9 of Figure 2.1, `main` does not block waiting for the call to `sayHello()` to return. The next statement is executed, and in this case since there is none, `main` terminates. Therefore, in Jade the end of computation is signaled by calling `CkExit()`, as shown in line 22 of the example program. CkExit gracefully terminates the program on all processors, ensuring that all messages have been delivered, flushing buffers, shutting down processes, etc.

Chares

A *chare* (line 6) is the basic parallel construct in Jade. It is essentially a migratable message-driven object. Its methods may be non-threaded or threaded, as described in Section 2.1.2. They can also be declared as `sync` which causes the calling method to block until the `sync` method returns.

In Jade, a chare is declared by specifying the `synchronized` attribute for the class, and by extending from the `chare` class, which implements the `Serializable` interface. In Java, the `synchronized` keyword applied to a method ensures that multiple threads sharing an object will access the method in mutually-exclusive fashion. The `synchronized` keyword cannot be applied to a class in Java; but in Jade, we use this notation to highlight the exclusive nature of execution of `chare` methods, as explained below. Thus:

```
1  public synchronized class MyChare extends Chare {
2    public int main(String []args) {...}
3    MyChare(){...}
4    public void memberMethod(int aParam) {...}
5    private AClass memberVar;
6  }
```

When a method of the chare is invoked, it executes to completion without interruption. This implies that only one method of a chare can execute at a time, i.e. in the Java sense it is as if all methods are `synchronized`. Of course, a `chare`'s methods may be `threaded`, which means that the synchronized behavior no longer holds.

Jade does not assume a shared memory model. A chare can have member variables, which can be primitive types, instances of sequential classes, or references to other parallel objects. However, its member variables are not permitted to be public. This is because a chare is a parallel object and an accessing method from another object cannot assume that the chare lives on the same processor. We considered permitting public members and automatically translating accesses into blocking method invocations, but decided against it because of the hidden cost of the access, the forced blocking and race conditions.

Member functions (which, of course, can include access methods) can be public, since they are a mechanism for information exchange. All public methods of a chare are invoked by asynchronous method invocation, and we use the term `entry method` to refer to the public methods of a chare as distinguished from the public methods of a sequential class. The return type of methods need not be `void`, but when invoked asynchronously, the return value is discarded.

ChareArrays

Jade supports 1D, 2D and 3D arrays of parallel objects as its primary parallel construct. Support for user-defined index types is part of the language, but is not yet implemented. The arrays can be sparse and created lazily, i.e. each element of an array is only created when first accessed, or all at once, as in the example program. Each element of the array is essentially a chare. The elements of the array are mapped by the system to the available processors.

A parallel array is declared by making the class `synchronized` and extending `ChareArray1D`, `2D` or `3D`. Line 15 of Figure 2.1 shows the declaration of a 1D `ChareArray`. Line 8 shows the instantiation of a `ChareArray` containing five elements. (The language supports `ChareArray` constructors with arguments, as described in the language manual.) The next line invokes the sayHello method of the first element (index value 0) of the `ChareArray`. The `int thisIndex` member of `ChareArray1D` (line 19) returns the index of the current element,

and the `CProxy thisProxy` member returns a proxy to the array itself. Broadcasting a method invocation to all elements of an array is accomplished by leaving out the index, e.g. `A.sayHello()`.

Threads

In Jade, threads of execution exist within an object – an object's methods may be marked as `threaded`, which means they execute in a user-level thread that can block. A non-preemptible, user-level thread is automatically created when a threaded method is invoked. The thread can call yield to block in the middle of the method. The thread ends once the function ends.

Migrating a thread would require migrating the parallel object containing it along with all other threads in the parallel object, since an object resides on only one processor. This general form of thread migration is currently not supported in Charm++ or Jade. Thread migration of special `TCharm` objects has been implemented in Charm++ and Jade. Any parallel object that inherits from the `TCharm` class and has only one thread of execution can be migrated.

`readonly` Variables

Jade also supports Charm++ `readonly` variables. These information sharing abstractions can be initialized in a `main` function, and are then broadcast to all processors. They cannot be modified once `main` ends.

In Charm++, `readonly`'s are defined globally. But in Jade, following the Java convention, they must be defined within a class. Internally, the Jade compiler makes `readonly`'s global after mangling names to ensure uniqueness.

MSA Arrays

The new MSA array entity provides a disciplined form of shared-memory programming to virtualization-based applications. From the user viewpoint, an MSA is a shared array of data elements. However, the array cannot be accessed in general read-write mode. It may be accessed in four currently supported modes: `read-only`, `write-many`, `accumulate`, and `owner-computes`. During a given phase of execution of the program, the array must be accessed in a single mode by all the parallel entities using it. Phases are separated by a `sync` call. MSA arrays are described in more detail in Chapter 3.

2.1.4 Realizing Various Programming Models in Jade

We have explained the general virtualized object programming model, the communication model and the parallel entities available in Jade. Now we describe how the pure message-driven object-oriented model, the thread-based object-oriented programming model, the message-passing model and the specifically shared memory model are realized in Jade.

Message-driven Programming Model

The pure message driven programming model does not use any threaded methods. Information exchange is by the use of asynchronous remote method invocation on non-blocking methods. Since blocking is not permitted in any object, there is no context-switching overhead: no stack or registers need to be saved and restored when switching execution from one object to another. Object migration is very quick and simple, only the class data needs to be packed and sent across processors. No stack transfer issues arise. Since only one method of a parallel object can execute at a time, no locking of class data is needed.

Latency tolerance is excellent, since programmers must break code down into units separated by communication dependencies, i.e. the natural chain of dependencies is expressed. Programming in this model is different from the traditional blocking-function-call model

of most programming languages. Programmers accustomed to blocking-call models find a substantial change of mindset is needed to successfully write message-driven programs.

Jade Threaded Programming Model

In traditional multi-threaded programming, a process creates threads of execution and manages them. Typically, one process is created per processor. Kernel level threads allow threads to block for I/O, etc., without blocking the process. For user-level threads, the programmer must use `yield` to manage latency.

In Jade, user-level threads are associated with methods of a parallel object, as described above. They come into existence when the method is invoked, and terminate when it completes. Threads are created in Jade by invoking threaded methods. In the Jade virtualization model a large number of work objects are created, usually there are several times as many objects as processors present, each of which may have several threads which are created in an on-demand fashion. When one thread calls `yield` (most Jade system calls call `yield`), another thread is scheduled for execution, thereby providing latency tolerance.

Message-passing Programming Model

In the traditional message-passing (MP) programming model, the program is a set of communicating processes. There is one process per processor. Messages are sent and received synchronously: i.e. both caller and callee block when they reach the send/receive, the message is exchanged and execution then continues. Non-blocking sends and receives are also provided. Within an MP process user-level threads can be used for parallelism.

AMPI [12] realizes the MP model using Charm++ and its runtime system. AMPI makes use of user-level threads and associates a single thread with each object to implement the MPI programming model. An MPI process transparently becomes a threaded method in a Charm++ object.

In Jade we support full MPI by translating to AMPI. Objects that inherit from the

`TCharm` class become AMPI "processes". AMPI system calls to send messages, obtain the processor rank, etc. are also available in Jade.

Specifically Shared Memory Model

The access modes of the MSA abstraction in Jade (see Section 2.1.3) are general enough to capture the majority of shared memory access patterns although MSA does not support a general `read-write` access mode. MSA coexists with the programming paradigms described above. Thus a set of chares and/or threads and/or AMPI "processes" can share an MSA array.

A Note on Multi-paradigm Programming

The described programming models can all co-exist in the same or different modules of a Jade program. Thus one class may be MD, another threaded, a third AMPI, a fourth can be an MSA array, and they can all call methods of each other within the restrictions mentioned. The models may also be mixed, e.g. a thread may use asynchronous method invocation and AMPI methods.

2.2 Compiler-supported Productivity Enhancements

Jade brings a compiler-based approach to runtime-based processor virtualization (PV) technology. This significantly enhances the ease of programming when compared to existing PV languages. Several researchers have argued for the unique productivity enhancements of combined compiler and runtime support[27, 3]. In this section, we describe some of the key productivity-enhancing features added by Jade. [2]

[2]Our approach has been to implement a feature directly in Charm++ as far as possible, in order to make it available to Charm++ programmers. We then design the feature in the Jade language at a higher level and translate Jade to Charm++. Some of the features listed may therefore also be available in Charm++.

2.2.1 Proxies, Parameter Marshalling, Parameter Passing

Proxies and Parameter Marshalling

The core Charm++ system defines a special data type called a `Message` and works only with `Message`'s in communication. Thus, for example, `entry method`'s can accept only one parameter which must be a `Message`. Reductions took a `Message`, not integers or floats. The user needed to define a `Message` for each kind of communication, register pack and unpack methods that would pack and unpack the data to the struct, and manage the creation and destruction of `Message` objects.

In addition to its payload data, a `Message` also carries a tag identifying the object and method it is directed at. This raw interface was what Charm++ programmers had to work with. Method invocation required calling a `CkSend` function that would actually tag the message and perform the invocation. Method invocation actually looked something like MPI_Send.

To simplify Charm++, we designed a proxy mechanism for parallel objects.[3] The proxy is a compiler generated wrapper class for each parallel class. The actual parallel object resides on an arbitrary processor. "Creating" a parallel object actually instantiates a proxy object, which fires off a request to the ARTS to actually create the parallel object. The ARTS returns a "handle", which is a globally unique ID, for the parallel object. This handle is stored in a member variable of the proxy object.

The proxy mechanism offers several advantages. Given the signature of an `entry method`, the Charm++ translator generates a method in the proxy class that does parameter marshalling for most cases: packing and unpacking of primitive data types, simple self-contained objects (without dynamic data), and arrays of the above. The translator defines a new message type, and in the proxy method packs the data by simple byte copy, and performs the

[3]The modern Charm++ translator is a rewritten and enhanced translator that incorporates and extends the proxy and parameter marshalling design of the translator that I implemented. The modern Charm++ translator was mostly implemented by Orion Lawlor and Milind Bhandarkar with enhancements by other members of the PPL lab.

message send to the actual parallel object.

Thus, `Message` mostly disappears from user code. Method invocations look like method invocations.

Parallel objects in Jade are actually local proxies that point to the actual parallel object. Thus

```
chare A {}
class B {
  public void f() {
    A a; // creates a local proxy to a
    a = new A(); // creates the real
                 // parallel object in lazy fashion.
  }
}
```

Proxies can be sent as parameters. They are passed by-value, which means the parallel object they contain is passed by reference-by-value (see discussion below).

A Discussion of Parameter Passing

The terms to describe assignment and parameter passing (i.e. the semantics in terms of l-values and r-values) in C++ and Java are often not clearly understood. We therefore first specify the meaning of the terms we use and then describe the kinds of data types in Jade and the semantics of assignment and parameter passing.

C only has pass-by-value. References are implemented by taking the address of a variable and passing the address by value. Hence:

```
#include <stdio.h>

void
swap(int *i, int *j)
{
  int tmp = i;
  *i = *j;
  *j = tmp;
}

main()
{
  int p = 1;
```

```
  int q = 2;
  swap(&p, &q);
  printf(``%d %d\n'', p, q); // prints "2 1"
}
```

C++ added pass by reference. The l-value of the formal parameter (i) is set to the l-value of the actual parameter (p). Changing i actually changes the r-value that p refers to.

```
#include <stdio.h>

void
swap(int &i, int &j)
{
  int tmp = i;
  i = j;
  j = tmp;
}

main()
{
  int p = 1;
  int q = 2;
  swap(p, q);
  printf(``%d %d\n'', p, q); // prints "2 1"
}
```

In Java, the claim is often made that primitive types are passed by value and objects by reference. But this is incorrect. All Java parameters are passed by value[63].

The confusion arises because object variables in Java are actually reference variables; i.e. "Object o" in Java means that "o" contains a reference to the actual object, it is not the object. Thus, in Java:

```
Object a = new Object(); // a refers to a new object on the heap
Object b;  // creates a reference to b, but does not allocate b, b is null
b = a; // b and a refer to the same object.
Object c = b.clone();  // c refers to a different object
```

Whereas in C++:

```
Object *a = new Object(); // a is allocated on heap
Object b; // constructor called, b is a new object on stack
b = *a; // copy constructor called.
       // b and a refer to different objects.
Object &c = b; // c refers to same object as b
```

In Java, when "Object o" is passed as a parameter, the reference is copied by value. The so-called pass by reference in Java is actually pass the reference by value. Thus, a working "swap" function cannot be implemented in Java. The following program prints "1 2".

```
class testpassbyref {
  public static void swap(Integer i, Integer j)
  {
    Integer tmp = i;
    i = j;
    j = tmp;
  }

  public static void main(String[] args) {
    Integer p = new Integer(1);
    Integer q = new Integer(2);
    swap(p,q);
    System.out.println(p + `` `` + q); // prints "1 2"
  }
}
```

In C++ terms, it is as if we are passing in the addresses by value, and modifying the variable containing the address. Thus:

```
void
swap(int *i_ptr, int *j_ptr)
{
  int *tmp = i_ptr;
  i_ptr = j_ptr;
  j_ptr = tmp;
}
```

Modifying i_ptr and j_ptr does not modify the rvalues of the actual parameters. We use the term pass-reference-by-value to describe the Java object parameter passing mechanism.

Now that the terminology is clarified, we discuss the actual semantics of Jade.

Jade Data Types and Semantics

Data types are categorized into three kinds in Jade. Primitive types include the standard built-in types: integer, float, double and so on. But unlike Java, the sizes of the primitive types are machine-specific. The second kind of data is sequential object data which are

```
1  // Jade
2  class A {
3    public void f(Object o){...};
4  };
5
6  class B {...}
7
8  class C {
9    public void g() {
10     A a;  //  creates a reference to a, but does not allocate it.
11     a = new A();  // allocates a on the heap
12     A a2;
13     a2 = a;  // a and a2 refer to the same object
14
15     B b = new B();
16     a.f(b); // b is passed by reference to sequential class A
17   }
18 }
```

Figure 2.2: Illustrative Jade code for sequential object parameter passing.

basically all classes other than the parallel classes. The third kind of data is the parallel objects such as `chares`, `ChareArrays`, `readonly` variables and MSA's.

Primitive objects are assigned and passed by value, just as in Java and C++.

Non-primitive sequential objects are assigned-by-reference, passed by reference-value to sequential objects, and passed by value to parallel objects. Thus, sequential behavior of sequential objects is the same as in Java. It is just that when they are passed to a possibly remote parallel object, it does not make sense to pass them by reference, since the reference might not be valid on a different processor. The intention is most likely to send over a copy of the data.

As an illustrative example, consider the code in Figure 2.2. On line 16 `b` is passed by reference when invoking the sequential object `a` which resides on the same processor. But when a sequential object is passed to a parallel object (which may reside on another processor), it is passed-by-value. So if `A` were a `chare` in the above example, a copy of `b` would be sent to it.

Finally, parallel objects are assigned-by-reference and passed-by-reference-value. Thus,

if B were a parallel object, it is actually a local proxy to the real parallel object and the proxy would be copied over. Of course, since the proxy contains the handle of the real parallel object, the copied proxy still refers to the same real parallel object, hence providing reference-value semantics.

In summary, primitive types are passed by value, parallel objects are passed by reference-value, and sequential objects are passed by reference-value to other sequential objects and by value to parallel objects.

2.2.2 Encapsulation and Automated Pack/Unpack Mechanism

Since a Jade parallel object cannot have public data members and since parameters are passed by value between parallel objects, we can conclude that the address space of a parallel object is private to it and cannot be directly accessed by any other parallel object. This tight encapsulation of data facilitates migration of objects.

Packing/unpacking of various language entities occurs in two situations: when the entity is passed as a parameter to a remote object, and when the entity or object containing the entity is migrated between processors. Both cases are handled by the same underlying PUP mechanism of Charm++.

Invoking a method of a parallel object can be either local or remote method invocation, depending on the location of the called chare (i.e. on which processor of the parallel system it is running).

Packing of sequential entities: Sequential object parameters are passed between parallel objects by copying them over in a message: this involves packing and unpacking of sequential objects. In Charm++, C++ struct-like byte copy works to pack and unpack (pup) most simple objects, but cannot handle C++ arrays, or objects containing dynamically-allocated data. In such cases, the programmer must define pack and unpack methods and register them with the Charm++ runtime system or must pack data into messages manually. [4]

[4]After the development of proxies described in a previous section, an object oriented pack/unpack method

Feature	Status	Details
primitive types	implemented	Object is copied over.
sequential classes	partial	Object is copied over. Support for circular references is not yet implemented.
sequential data arrays	implemented	Object is copied over.
references	implemented	The data referenced is copied over.
parallel objects: `chare`'s, `ChareArray`'s, MSA's	implemented	A handle is copied over for the parameter case, the object is moved for migration.
functions (function pointers)	excluded	See discussion in main text.
threads	partial	Single thread per chare is migrated. Multiple threads are pending Charm++ support.

Table 2.1: Migration support in Jade.

Packing of parallel entities: `Chares`, `ChareArrays` and MSA's can also be sent as parameters; however they are not copied. Instead, their handles/proxies are actually passed over. So one might think that parallel objects do not need pack/unpack support; and that only their proxies/handles would need to support pup. However, parallel objects can be migrated by the ARTS to perform load-balancing, fault-tolerance, or checkpointing. As a result, parallel objects must support PUP. In Charm++, in order for a parallel object to be migratable, the programmer must define a `pup` method.

In Jade, pack/unpack methods are auto-generated for every object (sequential or parallel) by the Jade compiler. Thus objects can be packed into messages for the purposes of parameter-passing, migration, checkpointing, etc. without any effort from the programmer. The automatic management of data movement in Jade simplifies programming in virtualization-based systems and removes a source of arcane bugs. Table 2.1 summarizes the pack/unpack support in Jade for various language entities.

Jade's compiler approach also opens up opportunities for optimization. In Chapter 4 we

was developed[32] that simplified user-written pack/unpack even more.

describe how as an enhancement in Jade, only live data is packed when migrating objects.

Limitations of Automated PUP.

Multiply-referenced data structures: The assignment-by-reference semantics of Jade allow multiple variables to refer to the same data object. In fact, circular data structures can be encountered too. Although these cases are uncommon, while packing data such references must be identified and encoded, and restored during the unpack phase. This feature has not been implemented in the current version of Jade.

Function pointers: Both Jade and Java do not allow pointers. However, Java is able to pass function pointers by creating an interface that contains the generic function and then creating an anonymous inner class which implements the desired function, and passing the anonymous class as a parameter. Jade does not support inner classes at this time and so this workaround does not work. However, Jade supports the Charm++ `CkCallback` class, which encodes a call to a method in a parallel object, thereby allowing "function pointers" for parallel entities. However, at this time, Jade has no mechanism to support encoding of calls to methods of sequential objects.

2.2.3 Multi-dimensional Data Arrays and Array Sections

At the outset, let us clarify that there are several kinds of "arrays" in Jade. `ChareArrays` are arrays of parallel objects. MSA's (Chapter 3) are a specifically shared parallel data array. And, of course, we have sequential data arrays which come in several flavors.

Java provides one dimensional data arrays. Higher dimensionality arrays are realized by creating arrays of arrays. These arrays are not efficient for numerical computations as discussed in detail in Artigas et al.[6, 48].

A quick way to implement multi-dimensional data arrays is to build on C++ STL vectors, but performance of our Jade Jacobi program using these arrays was significantly slower than the corresponding Charm++ program. We translated multi-dimensional Jade data arrays to C++ array-of-arrays with significant performance improvements. Seeking further

improvement, we then implemented block allocated 2D arrays instead of the Java-style array-of-arrays. Performance improved further due to the single-step address calculation and lookup.

However, some issues still remained from the Jade language design viewpoint. C++ arrays do not know their own size or dimensionality, which is required to pack/unpack them. Out-of-bounds indexing cannot be checked at runtime (in debug mode). And we wished to add the convenient HPF-like section notation to the Jade language, to reduce the necessity for the user need to copy data into temporary arrays for the purposes of parameter passing. Therefore we implemented a templated Jade sequential array library, `JArray`, and translate all array creation and access into calls to `JArray` methods. Through careful coding, the performance results we obtained were substantially similar to C++ arrays, as shown in Section 2.6.

Thus Jade supports Java-like arrays that know their size and can perform index checking. Jade also supports HPF-like `start:end:increment` array section notation. These features enhance productivity while maintaining performance comparable to sequential C++ arrays.

2.2.4 Reductions

Reductions in Charm++ have a difficult API. They take a pointer to the data to be reduced. The data each participant contributes can be an array, and so the number of elements must be given as a parameter. The type of the data and the reduction operation are specified using `enum`s. A call-back method must be given; at the end of the reduction, it is called with a `Message` parameter containing the result.

In Jade we simplify reductions significantly. Jade knows the data type, only the operation must be specified. The call-back must still be given, but it can expect a parameter of the same data type containing the result.

Internally, the Jade compiler defines a new call-back method which takes a `Message` parameter, extracts the data and calls the user's call-back with the data as a parameter.

2.3 Translating Java features into C++

In contrast to Java's run-time compilation, the Jade source code is translated to Charm++ source code, which is then *compiled* to binary/object code and executed on the target machine. This retains the performance benefits of C++ while providing the user with a clean programming model and enhancing productivity. The resulting Charm++ code scales well to large number of processors, as shown in Section 2.6. Java's standard libraries are not supported. We next discuss packages, static initialization and garbage collection in Jade.

2.3.1 package

Java's packages provide a unit of modularity. `package` members have greater visibility and access to data members of other classes within the same package. Java enforces a rigid runtime relationship between package names and directory structure. Thus, `class A.B.C` must be in file `A/B/C.java` and in `package A.B`.

Jade is compiled and symbols are resolved at link time. There is no need for a runtime relationship between package names and directories. No directory structure is enforced.

A Jade package corresponds to a module. All Jade files must be specified to be in some package. Jade source files (`A.jade`) can contain multiple classes: `A, B, C`. Package hierarchy is not supported. Thus, the fully qualified name of a Jade class is just `P.C`.

If file `A.jade` is part of `package P`, the Jade compiler translates its code into files P.h, P.ci and P.C.

`package P2` can use `package P` by using an import statement. `import P;` is translated to `#include P.h` and an `extern` declaration is added to the `P2.ci` file. This makes all classes and `readonly` variables in `package P` available to `P2`.

Java `packages` and package-private classes are supported by generating preprocessor conditionals:

```
1 P.h
2 #ifdef IN_PACKAGE_P
```

```
// package private class
#endif

P.C
#define IN_PACKAGE_P
#include ‘‘P.h’’
```

Coexistence of Jade and Charm ++

Jade code can use entities written in Charm ++ by declaring them as `extern` or using `#include` on their `.h` file. Similarly, Charm++ code can access Jade entities by including the package header file.

2.3.2 Static Data and Initialization of Class Members

Java allows anonymous static code in classes, and also allows initialization of class data members at the point of declaration in the class. Thus,

```
class A {
  static {
    // code here
    ALPHA
  }

  Object o = new Object();
}
```

In Java, anonymous static code is executed as class load time. As there is no class load in Jade because it is a compiled language, the semantics of anonymous static initialization code are changed to be called once per instance. This is useful for code that should be called from all constructors.

Initialization of non-static class data at declaration is not supported in Charm++/C++. The solution we designed generates an `_init()` method in every class and inserts a guarded call to it in every constructor. The guard is needed because in Java and Jade, one constructor of an object can call other constructors of the same object, but `_init()` must be called only once. The above code becomes:

```
1 class A
2 {
3   bool _called_init;
4   _init() {
5     ALPHA
6     o = new Object();
7   }
8 }
```

An interesting point to note is that `\_called\_init` cannot be implemented as a static variable within the _init method. When the object migrates, internal static variables are not packed and so their value is lost. This will cause `_init()` to be called again when the object is instantiated on the remote processor. Such internal static variables cannot even be packed because they are not accessible outside their containing block.

2.3.3 Garbage Collection

Garbage collection(GC) can be implemented in Jade using both the tracing (mark-sweep) and reference count approaches[68]. As described in Section 2.2.1, there is a clear separation in Jade between sequential data and parallel entities. Section 2.2.2 describes how no parallel object in Jade can contain a reference to any object within another parallel object. Each parallel object is a tightly encapsulated, self-contained entity. GC can therefore be performed at two levels: within a single parallel object to determine which sequential data is live, and across the program to determine which parallel objects are live.

The quickest approach to handle sequential data within a parallel object appears to be to support the Boehm-Demers-Weiser(BDW) conservative garbage collector[16]. BDW supports multi-threaded C++ code, which is required for multi-threaded Jade objects, with some restrictions.

Garbage collection of parallel objects requires a multi-processor parallel approach to analyze whether all proxies to the object are dead. Reference counting appears to be a more suitable mechanism for this analysis, especially since it can be easily added to the compiler

generated proxies. `ChareArrays` will need to be collected as a whole, because the proxy to any element of the `ChareArray` can be obtained from a proxy to the base object.

Our preliminary implementation of GC in Jade successfully limits the garbage collected heap size for sequential objects, but resulted in a segmentation fault for parallel objects. `delete` is supported in the interim.

An issue with garbage collection in general is that it depends on ensuring that references to unwanted data are removed from the program, but often programmers do not remove these references, e.g. from class variables. Therefore, GC could benefit from liveness analysis, which would help to identify dead data, even if a reference to it remains in the program. Chapter 4 discusses some issues with liveness analysis in Jade.

2.3.4 Comparison with Java

Given the large amount of research into translation of (sequential) Java to C, we decided to focus our efforts on the parallel features rather than on implementing all Java features.

Differences:

- Java supports dynamic loading of classes based on the `CLASSPATH`. However, our compilation product is a single executable and dynamic loading is not supported. It is possible to implement this capability in C++, and if applications require it, this feature can be added to Jade.

- Multiple classes can be defined in one Jade source file, unlike Java which has a very tight tying of classes to the naming structure to simplify dynamic class loading.

- Jade treats `main` differently from Java as described in Section 2.1.3.

- Java bytecode is portable across heterogeneous systems, while Jade code is compiled to a particular architecture. Java primitive types are standard across all platforms, but Jade types depend on the machine and C++ compiler used.

- Java threads have been replaced by Charm++ threaded methods.

Restrictions: Java names can be Unicode, whereas we support only ASCII. We do not currently support the standard Java runtime and libraries, nor do we support exceptions.

Detailed List of Differences

Table 2.2 shows the level of support in Jade for various features of Java. Some Java language features are supported exactly as in Java, others have a modified meaning, and some features are unsupported. Features in the latter class are usually not supported because we decided not to support them for lack of the resources to implement and maintain them; cases where we deliberately excluded a feature for some reason are identified as such. We use the keywords "supported", "partial"(ly supported), "modified", "unimplemented" and "excluded" to refer to the various kinds of features.

2.4 Translation Issues

The Jade compiler is implemented using the ANTLR parsing system which supports Java as an implementation language. For our grammar, we started out with a public domain ANTLR Java grammar[47].

A benefit of ANTLR over the *de facto* standard *yacc*, is that it can optionally generate an abstract syntax tree (AST) automatically. This feature was useful to get a quick start. However, looking back now, we feel that construction of a specialized AST as opposed to the generic AST generated by ANTLR is preferred for several reasons: the tree can be made smaller than the default construction, specialized nodes can hold specialized information, etc. Although ANTLR has features that permit the construction of specialized AST's and nodes, using them makes moot the benefit of automatic generation of AST's. The deciding benefit that determined our choice of ANTLR over *yacc* was that it supports Java as an implementation language. At the time we started implementation of Jade, options such as *byacc* that work with Java were not available.

Java Feature	Jade Status	Details
package	modified	See Section 2.3.1
import	supported	becomes `#include`
class, extends, implements	supported	uses multiple inheritance
nested class	unimplemented	
interface	supported	abstract class
nested interface	unimplemented	
primitive types: boolean, byte, char	modified	converted to 1-byte char, no unicode support
short, int, float, long, double	supported	architecture dependent
objects	supported	
arrays	supported	Converted to templated array that is PUP capable. See Section 2.2.3.
references	supported	See Section 2.2.1
templates	partial	Not all cases are handled, just enough for basic MSA's
identifier (a.b.c)	modified	Only 3-level names (package.class.name) are supported
identifier star (a.*)	unimplemented	
modifiers[5]: abstract, native, strictfp, transient, volatile	unimplemented	
public, private, protected	supported	for data fields, methods, constructors, class, and interface
static final	partial	implemented for data fields (constants)
synchronized	modified	added to class (for chare's, etc.), removed from methods
parameters	modified	See Section 2.2.1
"final" parameter	unimplemented	
try, throw, catch	unimplemented	
inner "static {}" class initializer, inner "{}" instance initializer	supported	See Section 2.3.2
constructor calls constructor, i.e. *this*() or *super*()	unimplemented	
initializers	supported	basic types, arrays
labels	supported	

Table 2.2: Comparison of Java features with Jade

Java Feature	Jade Status	Details
control statements: if, for, while, do, break, continue, return, switch	supported	
threads, run method, synchronized statements	excluded	replaced by Charm++ threaded methods
string handling	unimplemented	
true, false	supported	using `#define`'s
null	unimplemented	null objects have not been implemented
type.class	excluded	java.lang is unsupported
-	new	`threaded`, `blocking` and `readonly` keywords

Table 2.3: Comparison of Java features with Jade (continued)

However, ANTLR and related parser generators such as JavaCC and Sable CC, are very different from yacc. Antia and Breugel[5] provide an excellent overview of the issues and comparison of the different approaches to tree generation in ANTLR, JavaCC, SableCC and Zephyr ADSL.

The steps and byproducts involved in compiling a Jade source file are shown in Figure 2.3. The Jade source file (`.jade`) is translated into three files: the Charm++ interface file (`.ci`), a `.h` header file and a `.C` C++ source file

The `.ci` file is then fed to the Charm++ translator. It contains information about the parallel constructs in the program. The Charm++ translator generates `.decl.h` and `.def.h` files, which are `#include`'d in the `.h` and `.C` files respectively.

All the above generated files apart from the `.ci` file are then compiled with a C++ compiler and linked with the Charm++ and the Jade libraries to generate a single executable `a.out` file.

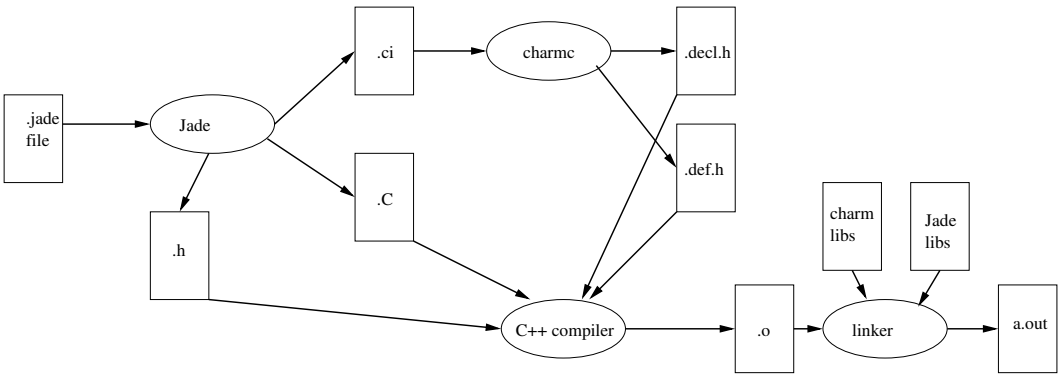

Figure 2.3: Compilation of a Jade source file.

Jade		Charm++	
JadeMatmul.jade	103	CharmMatmul.C	167
		CharmMatmul.h	41
		CharmMatmul.ci	20
Total	103	Total	228

Table 2.4: Number of lines of code for Jade and Charm++ matrix multiplication programs.

2.5 Productivity Results

Productivity improvements are hard to measure quantitatively. Number of lines of code (LOC) needed to implement the same algorithm are a rough measure that can be applied in certain cases. We provide LOC measurements for some Jade and Charm++ programs. We also point out how Jade programs are simpler to write and maintain. And we discuss various errors that Jade eliminates from Charm++ programs, and how Jade assists debugging. More Jade examples can be found in Section 3.2 of the MSA chapter.

2.5.1 Matrix Multiplication

Appendix A.1 shows a matrix multiplication program written in Jade. Table 2.4 shows the number of non-blank non-comment lines of code needed to implement the same matrix multiplication program in Jade and in Charm++. The Jade code is 55% smaller than the Charm++ version, and is contained in one file rather than three.

Jade		Charm++	
Jacobi.jade	162	Jacobi.C	179
		Jacobi.h	37
		Jacobi.ci	19
Total	162	Total	235

Table 2.5: Number of lines of code for Jade and Charm++ Jacobi programs.

2.5.2 Jacobi

Appendix A.2 shows a Jacobi program written in Jade. Table 2.5 shows the number of non-blank non-comment lines of code needed to implement the same Jacobi program in Jade and in Charm++. The Jade code is 31% smaller than the Charm++ version, and is contained in one file rather than three.

2.5.3 Jade Error Reduction and Debugging Assistance

The entire class of pointer-related memory management bugs is eliminated in Jade. Jade programmers do not encounter segfaults due to dereferencing invalid pointers.

Errors due to packing and unpacking data, especially for arrays and dynamically allocated data, are completely eliminated. Programmers do not need to follow complex packing rules: for instance, the description of `pup` in the Charm++ manual[53] occupies 8 pages plus a few more for system programmers. No `pup` methods or `pup` constructors need to be written in Jade.

Arrays in Jade are automatically packed for parameter passing and migration. Users do not need to pass in the size of the array as a parameter. Array section notation simplifies the specification of parts of arrays. When compiled in debug mode, the Jade compiler generates code that checks array indices for out-of-bounds conditions at runtime, which assists debugging tremendously in certain cases.

Users do not need to remember complex rules for passing parameters; for example, in Charm++, if a large object is passed as a parameter, the .ci file lists the parameter as

pass-by-value, but the .h and .C files should list it as pass-by-reference (i.e. prefixing the parameter with an &) for efficiency. This way, the data does not need to be copied over from the `Message` into the stack. Jade automates this. Beginner Charm++ programmers often make this mistake, which can lead to substantial performance degradation that is hard to identify.

The Jade language's single-file class definitions eliminate errors due to mismatched declarations in the .ci, .h and .C files. For example, in Charm++, failing to declare a variable as `readonly` in the .ci file will allow the code to compile and execute, but the `readonly` data will not be propagated to all processors. Such errors can be hard to catch.

Notation in Jade is consistent. In Charm++, a 1D `ChareArray` is indexed using array notation (`a1d[i]`), but a 2D `ChareArray` is indexed using a function call `a2d(i,j)`. In Jade all arrays (sequential, `ChareArrays` and MSA arrays) use array index notation.

The Jade type-checking pass catches errors such as declaring `chare` data members as `public`, which also catches attempts to directly access data members of other parallel objects. Jade detects attempts to modify `readonly` variables outside of `main`; these modifications will not be propagated across the system and can lead to subtle bugs.

At runtime, the MSA accesses are checked to ensure that all processes access the data in the same mode. Checks are also implemented to ensure that two worker objects do not write to the same element of the MSA array.

As further anecdotal evidence of productivity, I have written several Jade and Charm++ programs and now, even when I develop Charm++ programs, the ease of coding in Jade leads me to first write in Jade and use the translated code, bad indentation and all, as the basis for the equivalent Charm++ program.

Language	Execution Time (s)	Description
C++	12.186000	Sequential C++
Charm++	12.725498	Charm++ program using sequential C++ arrays
Jade	12.664641	Jade program using JArray's

Table 2.6: Jade and Charm++ matrix multiplication execution times on 2.4 GHz Pentium4 Linux workstation for problem size 2000*5000*300.

2.6 Performance Results

The objective of Jade is to retain the performance advantages of Charm++ while enhancing programming productivity. We demonstrate that Jade matches the performance of Charm++.

The computers used for our performance runs include a local Pentium4 cluster, the Lemieux cluster at the Pittsburgh Supercomputing Center, and the Tungsten cluster at the National Center for Supercomputing Applications. Our local cluster is a collection of single-cpu 2.4 GHz Pentium4 workstations running Linux and connected by 100 Mbps Ethernet. Each node of Lemieux contains four 1-GHz Compaq Alpha CPU's running Tru64 Unix; the nodes are connected by a high-speed Quadrix Elan interconnect. Each node of Tungsten contains two 32 bit Xeon 3.06 GHz processors with 3 GB memory; the nodes are interconnected by Myrinet.

2.6.1 Matrix Multiplication

We implemented a (non-MSA) matrix multiplication example in Jade, Charm++ and (sequential) C++. The outermost loop of the matrix multiplication was shared among the parallel objects. We created exactly as many worker objects as processors.

Table 2.6 shows the resulting execution time on a single-cpu 2.4 GHz Pentium4 workstation running Linux. The execution times of Charm++ and Jade are very close, and only very slightly slower than sequential C++.

We also studied performance on Lemieux. The results of our initial run are showed in

	Processors/Worker Threads
Language	1/1
Sequential C++	49.500000
Charm++	18.770541
Jade	48.829196

Table 2.7: Initial sequential matrix multiplication execution times on Lemieux for problem size 2000*5000*300.

Table 2.7. Examining the single-cpu results, we see that Charm++ significantly outperforms both Jade and sequential C++. The Jade matmul was taking 49 seconds, while the Charm++ one ran in 19 seconds. This behavior did not match the performance behavior on our local cluster (shown in Table 2.6).

We inserted timing calls to examine what was happening and determined that the matrix multiplication loop itself was significantly slower. Upon further analysis, we determined that the performance issue seen on Lemieux was that the `cxx` compiler was not optimizing the inner loop. The Jade-generated loop uses `JArray`'s instead of C++ arrays, and the generated code looks like this:

```
1   for(i = i0;(i < iN);(i++))
2     for(k = k0;(k < kN);(k++))
3       for(j = j0;(j < jN);(j++))
4         C(i, j) += (A(i,k) * B(k, j));
```

Instead of using array notation (e.g. `C[i][j]`), Jade code makes a function call (`C(i,j)`) for array access. The compilers on our local cluster were able to optimize the above code, and so Jade performed as well as Charm++ on them. But the Lemieux `cxx` compiler did not optimize the Jade code. When we changed the code in the inner loop to

```
1           C(i,j) += A(i,k) * (&B(k,0))[j];
```

the `cxx` compiler "got" it, and the Jade code ran slightly faster than the Charm++ code, as shown in Table 2.8. Results up to 4 processors are are shown in Table 2.9.

The speedup on a larger number of processors for a larger matrix multiplication (2048^3) is shown in Figure 2.4. These unoptimized results are shown merely to demonstrate that the

	Processors/Worker Threads
	1/1
Charm++	18.425529
Jade	16.616964

Table 2.8: Improved sequential matrix multiplication execution times on Lemieux for problem size 2000*5000*300.

	Processors/Worker Threads		
	1/1	2/2	4/4
Charm++	18.274148	9.953931	6.135394
Jade	16.055530	9.060960	4.738658

Table 2.9: Parallel matrix multiplication execution times on Lemieux for problem size 2000*5000*300.

scaling performance of Jade is essentially the same as Charm++. No significant additional overhead is introduced in sequential or parallel constructs, messaging, arrays, etc. Optimized matrix multiplication studies are deferred to Section 3.5 of the MSA chapter.

2.6.2 Jacobi

We implemented a Jacobi 2D decomposition example in Jade. Fig. 2.5 shows the speedup on Lemieux. On 100 processors, the speedup is about 80.

2.7 Related Work

Although Jade is based on java for syntax, it is not primarily intended as a parallel java project. Jade does not run all Java programs. It does it support Java libraries, the Java thread model, and so on, as discussed in Section 2.3.4.

Rather, Jade should be viewed as an enhancement to Charm++ and AMPI, and takes processor virtualization technology from the level of a library into the programming language level. The benefits that Jade provide are (a) improved programmer productivity through simplification of syntax, language design, and debugging support (b) current and potential

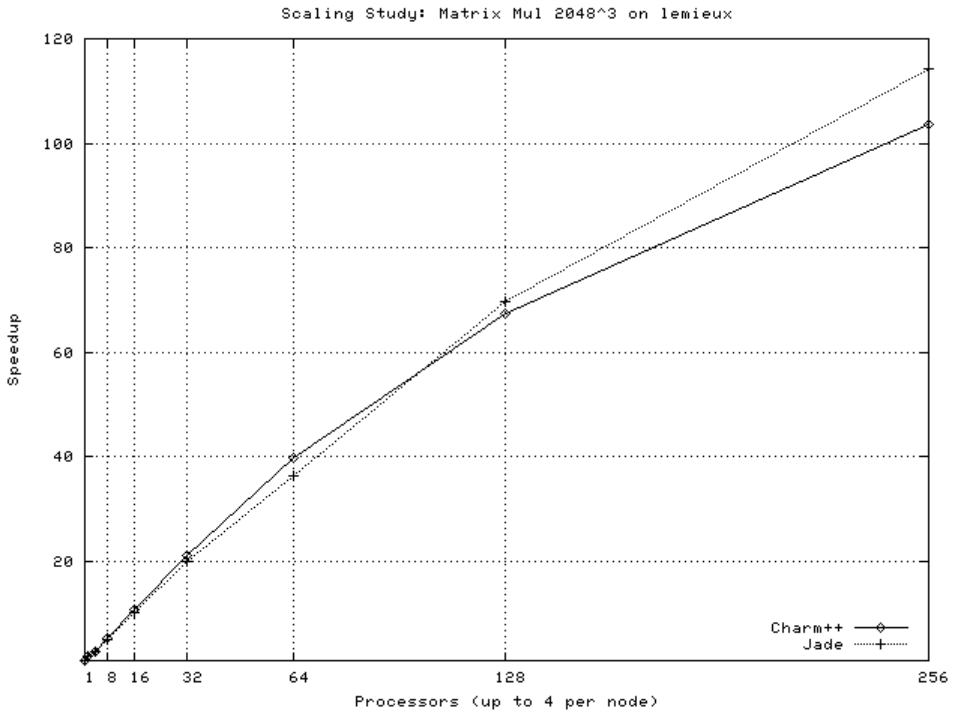

Figure 2.4: Scaling performance of Jade and Charm++ matrix multiplication.

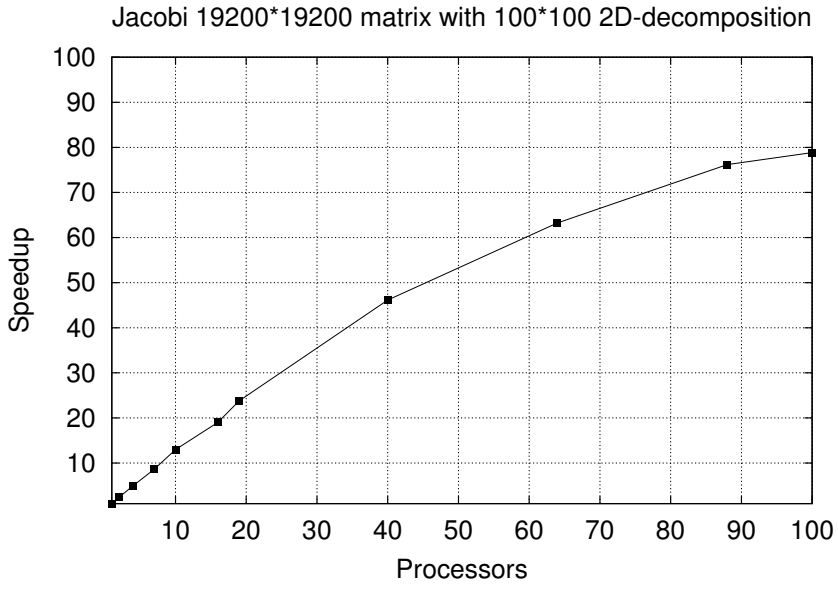

Figure 2.5: Scaling performance of Jacobi 2D written in Jade on Lemieux.

compiler optimizations.

As such, some of the key distinctions between Jade and any parallel java project are also Charm++ features: 1. processor virtualization i.e. creation of objects and threads that are not tied to the number of physical processors, automatic measurement-based load-balancing by an adaptive RTS through migration of objects. 2. multiple programming paradigms in the same program. 3. message-driven programming using super-lightweight chares with non-blocking methods that do not need to lock chare data and do not maintain stacks that need to be migrated.

Jade must still be compared with parallel java projects, since it looks like java and potential users may justly consider it an alternative to parallel java projects.

java's bytecode was designed for portability, but several java projects use it almost like a compiler intermediate format, analyzing and manipulating java byte-code instead of translating java source code. Bytecode is like getting compiler intermediate format (IF) for free one one end and getting an optimizing JIT compiler and runtime for free on the other end.

2.7.1 Parallel Java through RMI Enhancements

The Java language itself has built-in features to support parallelism: typically, Java threads are used for shared memory applications and either Java RMI or the standard socket interface is used for distributed applications. Standard Java RMI is slow, interpreted, requires explicit name binding, and provides no optimization for local objects[49].

The JavaParty[56] project takes an approach to parallel Java that is based on improving Java RMI. The execution time of a Java RMI call is analyzed and divided into time for low-level communication, serialization of parameters, and RMI overhead. Each of these issues is addressed by them with the ParaStation, UKA-Serialization[55], and KaRMI[49] projects, respectively. JavaParty code is interpreted, whereas Jade is compiled.

The Manta[7] project also works by improving Java RMI. It uses a compiler approach to generate specialized serializers, implements its own runtime and protocol to speed up RMI,

and can use a fast low-level communication layer (Panda/LFC) to improve speed. Manta supports runtime generation of marshalling code, distributed garbage collection, and compiler optimizations such as execution of non-blocking remote methods without a thread context switch. Manta interoperates with standard JVM and supports the Sun RMI protocol[43, 44].

Although JavaParty and Manta are minimally invasive to java programs, communication doth not a parallel language make. The level of programming remains that of MPI, using processes and threads for parallelism, and RMI for data exchange. There are no higher-level parallel entities such as global communication and synchronization, parallel arrays, shared data, etc. And of course, the key distinctions listed above apply, namely processor virtualization and automatic load-balancing, multiparadigm programming, and message-driven programming.

A previous version of parallel Java was implemented at the Parallel Programming Lab[35]. In that version, parallel constructs such as *chares* were added to Java. A JVM was executed on each processor of a parallel machine, and messages were injected into them using JNI and Java Reflection, i.e. the Java code was interpreted on a JVM. Parallel arrays, migration of objects, and global communication and synchronization were not supported.

2.7.2 Java Automatic Partitioning Tools

J-Orchestra[66], Addistant and Pangaea transparently allow Java objects to execute remotely instead of on the same processor. No changes to the original Java program are required. Methods invoked on remote objects are forwarded to them and the results sent back. Access to remote shared data is similarly handled. J-Orchestra supports object migration. Automatic load-balancing and migration are not provided, e.g. in J-Orchestra, the user must choose the placement of objects at the start of the program.

Coign[29] is designed for Microsoft's platforms and provides automatic partitioning of fine-grained COM objects. Coign works with binary COM code, first profiling the code to build a graph of the inter-component communication and then dividing up the code

for subsequent runs to minimize the communication. Coign has the significant limitation that all components that share the same data through pointers will be placed on the same processor. (Since everything in Java is accessed through references (implemented as pointers) the applicability of Coign technology to parallelization of Java is also very limited.)

The design intention of these automatic partitioning tools is for coarse-grained objects which do not share much data to be placed on different processors. They are not really applicable to fine or medium grained parallel applications with significant communication and/or load irregularity.

There is no support for higher level parallel language features, since the intention is to be Java-compatible.

2.7.3 Java DSM

The Java/DSM project [70] implements a JVM on top of the TreadMarks distributed memory system. The goal is to run multithreaded Java programs on multiple *heterogenous* processors using Java and DSM to solve the problems of running on heterogenous systems. Java provides the same programming model on heterogenous systems, and the addition of DSM removes the need for using inefficient and complicated Java RMI for remote communication and hides the disributed nature of the system from the programmer. The significant scaling issues of DSM are addressed in Section 3.4.

2.7.4 Titanium

Titanium[69] is a language-level parallel Java. The titanium compiler translates Java into C. Titanium adds a parallel SPMD programming style with a global address space and multi-dimensional Titanium arrays to Java and is especially suited to grid-structured computations. Each processor on a distributed memory machine maintains its data in a local demesne, and variables can be declared to limit access to only the local demesne of data, or to have unre-

stricted global access. The Titanium memory model follows the release consistency model and is similar to DSM; processors participating in a synchronization operation propagate their changes to each other. Several compiler analyses are performed, including identifying references to objects on the local processor. Although Titanium is not based on processor-virtualization technology, we believe this analysis can be applied to Jade also. Barriers and data exchange operations are used for global synchronization. Titanium does not provide the rich functionality of virtualization-based languages such as multiple programming models, automatic load-balancing, checkpointing, out-of-core execution, fault-tolerance, etc.

Our approach differs from Titanium in that we introduce a different parallel programming style, the message-driven style, into Java. Jade also supports a shared memory paradigm, described in Section 3.4 of this thesis. We support multi-paradigm programming and processor virtualization (with object migration and automatic load-balancing). The titanium compiler appears to be further along than ours, and provides several analyses that we could possibly benefit from using.

2.7.5 Proxy object mechanism

SunRPC[65] defined a mechanism for client-server Remote Procedure Calls that required the generation of client and server stub code in the C programming language. The CORBA specification[52] describes an Interface Definition Language and the usage of object-oriented proxies to implement remote method invocation.

2.8 Summary

Jade uses a compiler approach to simplify processor virtualization based (PV) parallel programming. Several syntactic and usage issues with current PV languages were identified and are solved in Jade without introducing performance overhead. These include parameter marshalling, automated packing and unpacking of data, improved sequential arrays,

type-checking of parallel types, and single-file class definitions. Features that enhance the productivity of parallel programming, both preventative (such as the absence of pointers) and reactive (such as array index checking in debug mode), were incorporated into the language. A multi-pass compiler framework that can also serve as the basis for further research was implemented. We presented a compiler-supported strip-mining optimization that improves MSA performance to the point of matching sequential array access. Example Jade programs and performance results were also presented.

The next chapter describes in more detail how Jade rounds out the multi-paradigm capabilities of PV systems by incorporating a disciplined shared address space programming model into the language.

Chapter 3

MSA: Multiphase Specifically Shared Arrays

Our experience with a number of parallel applications over the years indicates that there are distinct programming situations where SAS is an easier programming model whereas there are equally distinct situations where it is not. For instance, when there are data races, the shared memory paradigm, which allows for a much larger number of distinguishable interleavings of executions of multiple threads, tends to be a difficult paradigm. In contrast, a computation such as matrix multiply, where the data in input matrices is only read and data in the output matrix is only written, is relatively easier to express in SAS.

However, a general SAS model has not been implemented using processor virtualization because of performance issues. Shared address space (SAS) parallel programming models have faced difficulty scaling to large number of processors. The pure-hardware shared-memory solution using cache coherence, and the virtual memory page-management hardware-assisted relaxed consistency Distributed Shared Memory (DSM) realizations of shared memory both face significant scalability issues. The idea of distinguishing between access patterns of shared variables was studied in Munin[8] and also in Chare Kernel[24, 62] (the precursor to Charm[39, 36]) and is used in TreadMarks/NOW[40]. Optimized cache coherence mechanisms based on the specific access pattern of a shared variable show significant performance benefits over general DSM coherence protocols[18].

We suggest that the problems with SAS are due to trying to do everything (i.e. all kinds of information exchange between processes) with SAS. It may be more productive to incorporate SAS as a *part* of a programming model that allows private data for each thread, and mechanisms for synchronization and data-exchange such as message-passing or method-invocations. This frees us to support only those SAS access modes that can be efficiently and scalably supported on distributed memory machines including clusters, without being encumbered by having to support a "complete" shared-memory programming model.

Which access modes can be efficiently supported? `read-only` accesses, `write-many` accesses (where each location is updated by at most one thread), `accumulate` accesses (where multiple threads may update a single location, but only via a well-defined commutative associative operation), and an `owner-computes` mode seem to be the obvious candidates. These modes are general enough to capture the majority of shared memory access patterns. For all other operations not covered, one is free to use message passing or asynchronous message invocation, for example.

Another observation stemming from our application studies is that the access pattern for the same data changes distinctly between computation phases. For example, a matrix multiply operation (C = AxB) may calculate a C matrix in Phase I of the computation (where A and B matrices are accessed in read-only manner, and C is written-only or accumulated), whereas in the phase II, C matrix may be used in a read-only manner while A and B may be updated. These phases may then iterate.

These two observations, specifically shared data and access in phases, guided the design of *multi-phase specifically shared arrays*(MSA). For each MSA array, The programmer specifies one of the above access modes and can change it between phases. The phases are separated by synchronizations such as a barrier (as in release consistency[40]).

The restricted set of operations simplifies the consistency protocol and associated memory traffic. Only `read-only` mode requires remote data to be fetched. `Write-many` and `accumulate` modes initialize and update local copies of a page, data is exchanged at the end

of the phase. No invalidations or updates are needed during the course of any phase, but only at the end of the phase. As a result, MSA scales extremely well to a large number of processors, as demonstrated in Section 3.5. MSA does not support a general `read-write` access mode. MSA coexists with the message-passing paradigm (MPI/AMPI) and the processor virtualization-based message-driven paradigm(Charm++).

One of the original motivations for this work was computations performed at initialization, where efficiency is less important, and coding productivity is the dominant consideration. However, it quickly became clear that the method is useful more broadly beyond initialization. Of course, such broad use requires more serious consideration of efficiency issues. With strip-mining and `prefetch` calls (see Section 3.1), we provide efficiency comparable to that of local array accesses. One of the costs of DSM systems is the long latency on "misses"[8, 9]. Processor virtualization techniques that we have been exploring in Charm++ and Adaptive MPI (AMPI)[28] allow many user-level (lightweight) threads per processor, which help tolerate such latencies, as seen in Section 3.5.

A distinctive feature of MSA arrays is that the unit of coherence (or "page" size) can be specified by the programmer, i.e. it is not tied to the hardware virtual memory page size as in DSM, or the processor cache line size as in hardware shared memory. Thus, an MSA array can specify a page size of one data element, while another may specify a page size as large as the row of a matrix or the entire data array. The user-defined page size granularity can lead to significant performance improvements over fixed page sizes, as shown in Section 3.5.

Furthermore, MSA supports variable sized data elements. So, for example, MSA's can support a hash table, where each element of the array is a linked list. Page updates transfer the entire element using a user-specified pack/unpack method. Generalized `accumulate` methods are supported in MSA, permitting set union, list appending, etc. The generalized notion of accumulate accesses (see Section 3.1.4) is relatively new, although compiler research (e.g. Polaris[14]) has often focused on identifying commutative-associative operations.

The MSA abstraction is implemented as a library in Charm++ and AMPI, and as a

language-level abstraction in Jade. In Jade, compiler support enables us to automatically perform optimizations which have to be done manually by MSA users (such as strip-mining). Jade compiler support also features notational conveniences, allowing MSA arrays to be accessed using array ("`[]`") notation instead of `get`, `set`, and `accumulate` API calls.

In the rest of this chapter, we present the MSA model, its implementation, programming examples and performance results.[1]

3.1 MSA Design

As mentioned above, the MSA abstraction is implemented as a C++ library in Charm++ and AMPI, and as a language-level abstraction in Jade. The notation used in the exposition below uses the Jade language. Places where Charm++ behavior differs are mentioned in notes. Appendix B shows Charm++ notation.

Conceptually, an MSA is an array of data elements that can be globally accessed in an MIMD program in one of several dynamically chosen global access modes and in units of a user-defined page size. The modes are not expected to be fixed for the lifetime of an array, but for a phase of execution of the program. MSA's are implemented as a templated, object-oriented Jade class. Currently 1D and 2D MSA arrays are supported.

3.1.1 MSA Variable-sized Elements

The elements of an MSA can be one of the standard built-in types such as `int`s or `float`s, or a user-defined class with certain operations defined on it. The number of elements in the MSA is specified in the constructor when the MSA is created. [2]

[1]I wish to acknowledge the work of Rahul Joshi on implementing an initial version of MSA, and Orion Lawlor for improving the MSA API and performance.

[2]Currently, an MSA cannot be resized once created, but this is not a fundamental restriction, because the programmer can allocate a larger-than-needed array. Since MSA pages, described in the Section 3.1.2, are created lazily, allocating an MSA that is larger than needed does not actually instantiate the unused pages. The technical issue with permitting resizing for 2D MSA arrays is that MSA pages would need to be remapped over the data.

In Charm++ (but not in Jade), for complicated element types (such as linked lists), a `pup()` method must be defined by the user for the element type: this is used to pack and unpack the element when pages are shipped around. (More details of the *PUP* framework can be found in [32].) PUP methods for standard types are predefined. The `pup` framework enables MSA elements to be a linked list or other variable sized data structure. Jade auto-generates PUP methods as described in Section 2.2.2, and so automatically supports variable-sized data types.

3.1.2 MSA User-specified Pages and Data Layout

Internally, the MSA array is split up into "pages" of a user-defined size. The page size is specified as a template parameter at the time the MSA is created. It is expressed as a number of elements and can range from a single element to the total number of elements in the array. For 2D MSA arrays, the data layout can also be specified at creation time. Currently, row-major and column-major data layouts are supported, and we plan to support block partitioned layout.

The array of "pages" is implemented as a `ChareArray` (Section 2.1.3) object. `ChareArrays` are managed by the Jade runtime system; they are capable of processor migration and can participate in system-wide load-balancing based on communication patterns, computational load, etc., can be automatically checkpointed and restarted on a different number of processors, and can be swapped out to disk for out-of-core execution when needed.

3.1.3 Page Replication in the Local Cache

The MSA runtime system on each processor fetches and replicates remote pages into local memory on demand, or instantiates blank local copies of remote pages when needed, based on the mode (described below) of the MSA. The amount of local memory so used (the *local cache*) on each processor can be constrained by the user for each MSA. When the local cache

fills up, pages are flushed out using a user-defined page replacement policy. Standard page replacement algorithms such as least-recently used(LRU), FIFO, etc. can be implemented. MSA implements an extremely efficient "not recently used" default policy that keeps track of a few most recently used pages in a circular list: any page not listed there can be flushed out.

3.1.4 MSA Modes and Operations

The MSA is globally accessed in one of several *access modes*, which can be changed dynamically over the life of the program. The mode of an MSA is set implicitly by the first operation (read, write, or accumulate) on it after a `sync` or `enroll`. Each phase of execution in a particular mode is terminated by running a `sync` operation on the array.

The modes supported are `read-only`, `write-many` and `accumulate`, and have been chosen for simplicity and efficient implementation, as described below. The modes avoid all race conditions, thereby eliminating the need for any coherence-related communication during the mode.

In the `read-only` mode, the elements of the array are read-only by the worker threads. `read-only` is efficiently implemented by replicating pages of the array on each processor on demand. Reading a page that is not available locally causes the accessing thread to block until the page is available. `read-only` mode is the only mode in which the accessing thread can block, `write-many` and `accumulate` modes never block. User-level or compiler-inserted `prefetch` calls can be used to improve `read-only` performance. Since the page is read-only, no invalidates or updates need to be propagated. This makes `read-only` mode very efficient.

In the `write-many` mode, several threads are permitted to write to the elements of an MSA, but to different elements, i.e. at most one thread is permitted to write to a particular element. `write-many` is efficiently implemented by instantiating a blank local copy of an accessed page on any processor that needs the page. No page data and no page invalidates or updates need to be communicated during the phase. At the end of the phase, the changed

data in the local cache are forwarded to the above-mentioned `ChareArray` page object, where they are merged.

In the `accumulate` mode, multiple threads can perform an accumulate operation on any given element. `accumulate` is implemented by accumulating the data into a local copy of the page on each processor, (instantiated with the identity element on first access), and combining these local values with the home page at the end of the phase. Once again, no page data or coherence traffic is transmitted during the phase.

In most cases, data in an array is accumulated using only one operation, i.e. addition, or multiplication; and so the `accumulate` operation to be used is specified at creation of the MSA. MSA provides templated $Null$, Add, $Product$, Max and Min accumulators. The `accumulate` operation can be changed at any time by invoking a method of the MSA: a `sync` is needed after the change so that the change can propagate to all processors. Accumulation using the common sum, product and max operations is provided for built-in types. For the general case, setting the `accumulate` operation involves passing in a class that contains `accumulate()` and `getIdentity()` methods. This allows the accumulate operation to be user-specified. In combination with the `pup` framework, this allows `accumulate` to handle complex operations such as set union, appending to a hash table in which each element is a linked list, and so on.

The user is responsible for correctness and coherence; e.g., if an array is in `write-many` mode, it is the user's responsibility to ensure that two processors do not write to the same location. The system assists by detecting errors at run-time.

Because of processor virtualization, the number of worker threads accessing an MSA object can be different from the number of processors the program is running on. The `sync` operation needs to know this number in order to determine when all threads have reported their changes. Furthermore, each MSA array maintains only one cache for page replication per processor so that pages are not replicated unnecessarily even if multiple threads accessing the same MSA array are present on the processor. (This feature is made feasible because the

nature of the various MSA modes avoids race conditions.) The distribution of the worker threads across the processors of the system needs to be known, in order to ensure that the data changed in a processor's cache is only sent after all *local* worker threads call their `sync` operation. This is achieved by requiring the worker threads accessing an MSA to participate in an `enroll` operation at startup so that the system can determine the number of worker threads on each processor.

In certain cases, it is desirable for the processor owning a page to perform some function on the elements of the page. Such an *owner-computes facility* is supported in MSA. The API for MSA requires the user to define a class that inherits from the `MSA_Page_Array` class, and pass this class to the MSA constructor. Thereafter, functions in this class can be called very simply by using `ChareArray` broadcast, which causes the function to execute on each page in the `MSA_Page_Array`.

3.1.5 MSA Optimizations

For the `write-many` and `accumulate` modes, the MSA runtime on each processor keeps track of which elements of the page are written. This can be done by initializing the page with an unused value when the page is created in the local cache; thereafter, any changed value can be easily identified. This works out well for elements such as `floats` or `doubles` since we can use `NaN` as a marker value, but MSA also provides a default alternate mechanism that keeps track of changed elements in a local bit-vector.

DSM uses page fault hardware to detect access to a page that needs to be fetched from a remote processor. This is a relatively efficient mechanism, but forces the page size to match the virtual memory page size, and does not permit a type-aware system that can support variable-sized data elements on the page. These features significantly affect efficiency and programming ease as discussed in Sections 3.5 and 3.2.3, and we wished to support them in MSA. Therefore, in MSA, when accessing data in `read-only` mode, the MSA system checks whether data is available in the local cache on every access. Since this is expensive,

```
1 for i=subsection of size N/P  // Rows of A matrix
2   for j=1..N                  // Columns of B matrix
3     for k=1..N
4       result += A[i][k] * B[k][j]
5     C[i][j] = result
```

Figure 3.1: Matrix Multiplication pseudocode: 1-level decomposition.

using compiler support in Jade we strip-mine *for* loops and use local (non-checked) accesses within a page. `write-many` and `accumulate` modes also benefit from such non-checked accesses, since it reduces one lookup when converting an index into an address. As shown in Section 3.5, strip-mined MSA performance matches the performance of sequential array access. Both `prefetch` and direct local access methods for strip-mining are provided in the library API for Charm++ programs.

3.2 Example Programs

In this section, we present several example programs to display the productivity benefits of the MSA abstraction and illustrate how the basic MSA modes suffice to provide shared address space programming for realistic applications.

3.2.1 Matrix Multiplication

The pseudocode for a matrix-matrix multiplication using a straightforward row-wise decomposition is shown in Figure 3.1 (assuming $N*N$ matrices and P processors). The i dimension is divided among the worker threads. Thus in this case, each thread will request a subset of the rows of A and C, and the entire B matrix.

Relevant sections of the corresponding MSA program are shown in Fig. 3.2. 2D MSA arrays are used in this example. We use row-major data layout for the A and C matrices, and column-major for the B matrix. The page size for each MSA is specified when defining

the MSA (lines 1–2). For our matrix problem size of 2000*5000*300, we set it to 5000 in order to fetch an entire row of A or C, or column of B. The page size is a crucial parameter for performance. (We could set B's page size to the total number of elements in B, and that would fetch the entire B matrix into local memory upon the first access to it. However, such a large data transfer would cause a delay at startup while B is fetched on every processor; we choose a smaller size so that the transfers can be overlapped with computation by other worker threads.) The per processor cache size is specified when instantiating the MSA (lines 5–7). We set the cache size to hold at least the working set of data, unless that is too large to fit in the available memory.

For the algorithm shown in Figure 3.1, if all matrices are $N * N$ matrices, the required number of elements required by each of P processors works out to $N^2 + 2N^2/P$, i.e. $O(N^2)$. The computation requires an add and a multiply operation inside three nested loops, which works out to $2N^3/P$ operations per processor. The computation/communication rato is $O(N/P)$.

We can improve this ratio by decomposing the inner loops of the matrix multiplication. Consider the pseudocode shown in Figure 3.3 for a block decomposition matrix product. The `i` and `j` loops are shared ampng the virtual processors. Here too, exactly one thread is responsible for each element of C and the `write-many` mode can be used for C. The number of elements required by each processor is $2N^2/\sqrt{P}$, since each thread needs $N/\sqrt{P}$ rows of A and $N/\sqrt{P}$ columns of B. The computation/communication ratio improves over the previous case to $O(N/\sqrt{P})$

Finally, consider the case where we decompose in the k-dimension as well, shown in Figure 3.4. Here, `C[i][j]` is written to by many threads and the MSA `accumulate` mode is useful. An *Add* accumulator class can be specified as the default when creating the C MSA. The number of elements required by each processor reduces from the previous case to $2N^2/P^{2/3}$. And the computation/communication ratio improves further over the previous cases, to $O(N/P^{1/3})$.

```
1  typedef MSA2D<double, MSA_NullA<double>, 5000,MSA_ROW_MAJOR> MSA2DRowMjr;
2  typedef MSA2D<double, MSA_SumA<double>, 5000,MSA_COL_MAJOR> MSA2DColMjr;
3
4  // One thread/process creates and broadcasts the MSA's
5  MSA2DRowMjr arr1(ROWS1, COLS1, NUMWORKERS, cacheSize1); // row major
6  MSA2DColMjr arr2(ROWS2, COLS2, NUMWORKERS, cacheSize2); // column major
7  MSA2DRowMjr prod(ROWS1, COLS2, NUMWORKERS, cacheSize3); //product matrix
8
9  // broadcast the above array handles to the worker threads.
10 ...
11
12 // Each thread does the following code
13 arr1.enroll(numWorkers); // barrier
14 arr2.enroll(numWorkers); // barrier
15 prod.enroll(numWorkers); // barrier
16
17 while(iterate)
18 {
19     for(unsigned int c = 0; c < COLS2; c++) {
20         // Each thread computes a subset of rows of product matrix
21         for(unsigned int r = rowStart; r <= rowEnd; r++) {
22             double result = 0.0;
23             for(unsigned int k = 0; k < cols1; k++)
24                 result += arr1[r][k] * arr2[k][c];
25
26             prod[r][c] = result;
27         }
28     }
29
30     prod.sync();
31     // use product matrix here
32 }
```

Figure 3.2: Jade MSA matrix multiplication code: 1-level decomposition.

```
1  for i= subsection of size N/sqrt(P) // Rows of A matrix
2    for j= subsection of size N/sqrt(P)  // Columns of B matrix
3      for k=1..N
4        result += A[i][k] * B[k][j]
5      C[i][j] = result
```

Figure 3.3: Matrix multiplication pseudocode: 2-level decomposition.

```
1 for i=subsection of size N/cuberoot{P}      // Rows of A matrix
2   for j=subsection of size N/cuberoot{P}    // Columns of B matrix
3     for k=subsection of size N/cuberoot{P}
4       C[i][j] += A[i][k] * B[k][j]
```

Figure 3.4: Matrix multiplication pseudocode: 3-level decomposition.

The simplicity of coding the above examples using the MSA abstraction is stressed. No message packing or communication needs to be specified to share subsections of A, B, and C. For Figure 3.4, a distributed-programming example would require an array reduction at the end of the loop to accumulate `C[i][j]` across the worker threads. With MSA, the reduction is automatically performed using a page as the unit of the reduction.

3.2.2 Molecular Dynamics

In classical molecular dynamics based on cut-off distance (without any bonds, for this example), forces between atoms are computed in each timestep. If two atoms are beyond a cutoff distance the force calculation is not done (to save computational cost, since force drops as a square of the distance). After adding forces due to all atoms within the cutoff radius, one calculates new positions for each atom using Newtonian mechanics.

The pseudocode for molecular dynamics using MSA is shown in Figure 3.5. The key data structures used are:

- `coords` [i]: a vector of coordinates (x,y,z values) for each atom i.
- `forces` [i]: a vector containing forces (x,y,z values) on atom i.
- `atomInfo` [i]: a struct/class with basic read-only information about each atom such as its mass and charge.
- `nbrList`: `nbrList` [i][j] is true if the two atoms are within a cutoff distance (neighbors).

There are three phases in each timestep. The `atomInfo` array is `read-only` in all phases. During the force computation phase, the `forces` array is `write-many` whereas the `coords` array is `read-only`; while during the integration phase, this is reversed. Every 8 steps (here) we recalculate the `nbrList` in phase III, where `nbrList` is `write-many` and `coords` is `read-only`. The code assumes a block partitioning of the force matrix as suggested by Hwang et al.[30] or Plimpton et al.[59].

3.2.3 FEM Graph Partitioning

As an example of the power of the generalized `accumulate` operation, we present in Figure 3.7, a part of a program that deals with an unstructured mesh for a finite-element method (FEM) computation. The data structures used are shown in Figure 3.6. The mesh connectivity data is available at input in the `EtoN` array: for each element `i`, the `EtoN` [i] contains three node numbers (we assume triangular elements). The objective is to produce the `EtoE` array, where `EtoE` [i] contains all element (numbers) that are neighbors of E_i. E_1 is said to be a neighbor of E_2 if they share a common node. So, e2 and e3 are neighbors because they share Node 4.

The algorithm for doing this using MSA proceeds in two phases. In the first phase, an intermediate array `NtoE` is created by accumulation: `NtoE` [j] contains all elements that have n_j as their node. To construct this, each thread processes a section of the `EtoN` array.

The `accumulate` operation in this example is user-defined set accumulation. It demonstrates the power of the MSA abstraction: allowing elements of the MSA array to be variable-sized. The user defines a class for the element, which contains a reference to an instance of the same class, and implements an assignment operator, a + = operator, and a `pup` method. (If using `Jade`, the `pup` is generated by the compiler. And since `Jade` does not support operator overloading, the user defines `assign` and `accumulate` methods.)

As a matter of efficiency, pages are normally copied as a string of bytes in `Jade`; but when using variable-sized pages, each element needs to be copied using the `pup` method. An API

```
1  // Declarations of the 3 arrays
2  class XYZ; // { double  x; double y; double z; }
3  typedef MSA1D<XYZ, MSA_SumA<XYZ>, DEFAULT_PAGE_SIZE> XyzMSA;
4  class AtomInfo;
5  typedef MSA1D<AtomInfo, MSA_SumA<AtomInfo>,
6                DEFAULT_PAGE_SIZE> AtomInfoMSA;
7  typedef MSA2D<int, MSA_NullA<int>,
8                DEFAULT_PAGE_SIZE, MSA_ROW_MAJOR> NeighborMSA;
9
10 XyzMSA coords;
11 XyzMSA forces;
12 AtomInfoMSA atominfo;
13 NeighborMSA nbrList;
14
15 //broadcast the above array handles to the worker threads.
16 ...
17
18 // Each thread does the following code
19 coords.enroll(numberOfWorkerThreads);
20 forces.enroll(numberOfWorkerThreads);
21 atominfo.enroll(numberOfWorkerThreads);
22 nbrList.enroll(numberOfWorkerThreads);
23
24 for  timestep = 0 to Tmax {
25   /*************** Phase I ***************/
26   // for a section of the interaction matrix
27   for i = i_start to i_end
28     for j = j_start to j_end
29       if (nbrList[i][j]) { // nbrList enters ReadOnly mode
30         force = calculateForce(coords[i], atominfo[i],
31                                coords[j], atominfo[j]);
32         forces[i] += force; // Accumulate mode
33         forces[j] += -force;
34       }
35   forces.sync();
36
37   /*************** Phase II ***************/
38   for k = myAtomsbegin to myAtomsEnd
39      coords[k] = integrate(atominfo[k], forces[k]); // WriteOnly mode
40   coords.sync();
41
42   /*************** Phase III ***************/
43   if  (timestep %8 == 0) { // update neighbor list every 8 steps
44     // update nbrList with a loop similar to the force loop above
45     ... nbrList[i][j] = value;
46
47     nbrList.sync();
48   }
49 }
```

Figure 3.5: Jade MSA molecular dynamics example code.

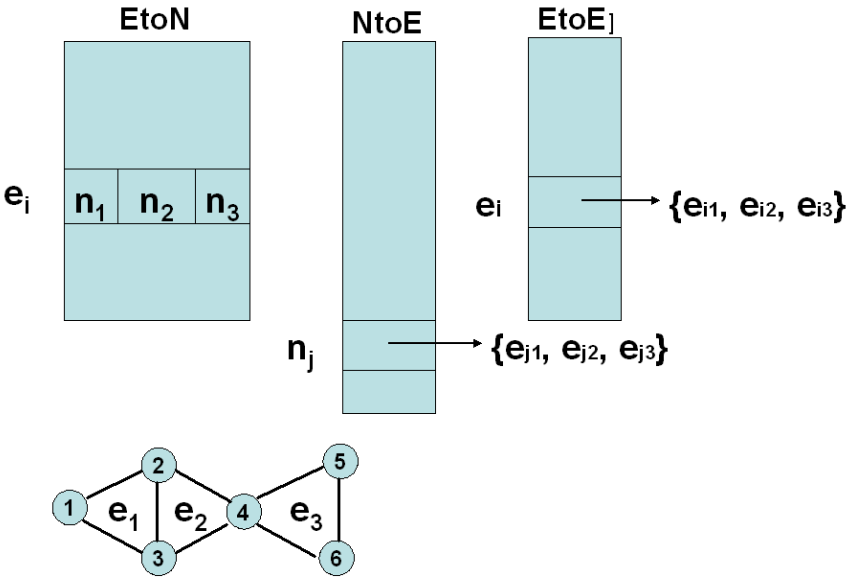

Figure 3.6: FEM graph partitioning data structures.

```
1  // Phase I: EtoN: RO, NtoE: Accu
2  for i=1 to EtoN.length()
3    for j=1 to EtoN[i].length()
4      n = EtoN[i][j];
5      NtoE[n] += i;  // set accumulate operation
6
7  // Phase II: NtoE: RO, EtoE: Accu
8  for j = my section of NtoE[j]
9    //foreach pair e1, e2 element-of NtoE[j]
10   for i1 = 1 to NtoE[j].length()
11     for i2 = i1 + 1 to NtoE[j].length()
12       e1 = NtoE[j][i1];
13       e2 = NtoE[j][i2];
14       EtoE[e1] += e2; // set accumulate
15       EtoE[e2] += e1; // set accumulate
```

Figure 3.7: Jade MSA FEM Code.

is provided for the user to over-ride the default byte-copy behavior when using variable-sized data elements.

3.2.4 Livermore Loops

The Livermore Loops[25] consist of a set of kernels of loops extracted from operational code at LLNL. The loop kernels display a wide range of loop access patterns and have been widely used as a performance benchmark.

We examined the set of 14 loops and analyzed whether they can be expressed using the MSA paradigm. For instance, consider Kernel 1:

```
1    for ( l=1 ; l<=loop ; l++ ) {
2        for ( k=0 ; k<n ; k++ ) {
3            x[k] = q + y[k]*( r*z[k+10] + t*z[k+11] );
4        }
5    }
```

Here, y and z can be placed in `read-only` mode, and x in `write-many` mode to implement this kernel in MSA.

But consider Kernel 11:

```
1    for ( l=1 ; l<=loop ; l++ ) {
2        x[0] = y[0];
3        for ( k=1 ; k<n ; k++ ) {
4            x[k] = x[k-1] + y[k];
5        }
6    }
```

Here, y can be placed in `read-only` mode, but the close dependence of $x[k]$ on $x[k-1]$ makes it difficult to express x using MSA's. Each iteration of the k loop would require a `sync` operation on x for correct operation. Of course, it should be pointed out that this *close dependence issue*, as we will call it in our discussion below, is difficult for any compiler to parallelize due to the nature of the dependences.

The details required to implement each kernel in the MSA paradigm are described below:

1. x in `write-many`, y and z in `read-only`

2. duplicate of x in `write-many`, x and v in `read-only`

3. x and z in `read-only`

4. duplicate of x in `write-many`, x and y in `read-only`

5. close dependence issue for x; y and z in `read-only`

6. close dependence issue for w; b in `read-only`

7. x in `write-many` mode, u, y, z in `read-only`

8. break innermost loop into two parts (1) du1, du2, du3 in `write-many`, u1, u2, u3 in `read-only` (2) du1, du2, du3 in `read-only`, duplicates of u1, u2, u3 in `write-many`, and u1, u2, u3 in `read-only`.

9. Since 2D MSA arrays cannot have different modes for each row, the px array must be re-written as separate 1D arrays for each row. Then in iteration i, px$i$ is placed in `write-many` mode, and all other arrays in `read-only` mode.

10. Basically, PX(i,j) is first read, then overwritten. Therefore, we use a duplicate PX in `write-many` mode and the original PX in `read-only` mode. At the end of the comutation, we overwrite the old PX.

11. close dependence issue for x; y in `read-only`. This loop is a prefix sum and can be expressed using a different algorithm that might be more amenable to MSA.

12. x in `write-many`, y in `read-only`

13. b, c, y, z in `read-only`; but close dependence issue for p

14. close dependence issues for vx, xx, ir, rx

3.3 Jade Strip-mining Compiler Optimization

Figure 3.8 shows a Jade loop that accesses an MSA array `arr1` using array index notation. The equivalentCharm++ loop (generated by the Jade compiler, or written by a Charm++ programmer using the MSA library API described in Chapter 3) is shown in Figure 3.9. The MSA `get` method, shown in Figure 3.10, is declared as `inline`, and this eliminates the function call overhead. However, the `get` function contains the overhead of an `if` statement in every call, which is expensive for performance.

An obvious optimization to try is to strip-mine the loop so that once a page is fetched into the local MSA cache, its elements are thereafter accessed directly using a pointer and offset notation, which completely bypasses the `get` function call. We implemented such an optimization in the Jade compiler; the generated code for the input loop of Figure 3.8 is shown in Figure 3.11.

Several cases can arise when fetching pages for local access:

1. page size == row size
2. page size < row size, and divides evenly
3. page size < row size, and does not divide evenly
4. page size > row size, and divides evenly
5. page size > row size, and does not divide evenly

`i0`, `iFinal`, and `iDelta` are the bounds and increment of the outer loop; and `j0`, `jFinal`, and `jDelta` are the same for the inner loop. The 2D MSA indices must be converted into a page/offset format to access the internal MSA array of pages. `indexSt` and `indexEnd` are the starting and ending internal-indices of the current page that has been fetched. The user-index tuple `(iSt,jSt)` represents the starting user-indices on the current page, and tuple `(iN,jN)` represents the ending user-indices on the current page. `jEnd` and `jFinal` are needed for the cases when the page size and row size do not divide evenly. The column-major case is symmetric with the row-major case: `ROWS` and `COLS` are interchanged, and `i` and `j` are interchanged.

```
1 for(int r = rowStart; r <= rowEnd; r++)
2     for(int c = 0; c <= TheMain.COLS1-1; c++)
3         arr1[r][c] = 1.0;
```

Figure 3.8: Jade MSA loop code input for strip-mining optimization.

```
1 for(int r = rowStart;(r <= rowEnd);(r++))
2   for(int c = 0;(c <= (TheMain::COLS1 - 1));(c++))
3     arr1.set(r, c) = 1.0;
```

Figure 3.9: Equivalent Charm++ MSA loop code.

The performance improvements due to the strip-mining optimization are detailed in Section 3.5.

3.4 Related Work

3.4.1 DSM, TreadMarks, and Munin

Distributed Shared Memory (DSM) is a much-studied software-level shared memory solution. Typically, DSM software uses the virtual memory page fault hardware to detect access to non-local data, which it then handles. It works at the page level, fetching and delivering virtual memory pages. DSM uses relaxed consistency memory models to reduce false sharing overheads and improve performance[31, 1].

Munin[8, 9] and TreadMarks[40] are DSM implementations. TreadMarks implements the *release consistency* memory model, which typically does not require any additional synchronization over a general shared memory (sequential consistency) program. To reduce false sharing overheads, their *multiple-writer* coherence protocol allows multiple threads to write to independent locations within a page.

Munin takes such coherence optimizations further, and identifies several access modes

```
1  /// Return a read-only copy of the element at idx.
2  ///   May block if the element is not already in the cache.
3  inline const ENTRY& get(unsigned int idx)
4  {
5    unsigned int page = idx / ENTRIES_PER_PAGE;
6    unsigned int offset = idx % ENTRIES_PER_PAGE;
7    return readablePage(page)[offset];
8  }
9
10 // MSA_CacheGroup::
11 inline const ENTRY_TYPE* readablePage(unsigned int page)
12 {
13   accessPage(page,Read_Fault);
14
15   return pageTable[page];
16 }
17
18 /// Make sure this page is accessible, faulting the page in if needed.
19 // MSA_CacheGroup::
20 inline void accessPage(unsigned int page,MSA_Page_Fault_t access)
21 {
22   if (pageTable[page] == 0) {
23     pageFault(page, access);
24   }
25 #ifndef CMK_OPTIMIZE
26     if (stateN(page)->state!=access) {
27       CkPrintf("page=%d mode=%d pagestate=%d",
28                page, access, stateN(page)->state);
29       CkAbort("MSA Runtime error: Attempting to access a page that\
30  is still in another mode.");
31     }
32 #endif
33   replacementPolicy->pageAccessed(page);
34 }
```

Figure 3.10: MSA `get` method code.

```
1  {
2    int accessMode = (-1);
3    int accessPattern = (-1);
4    int MAJOR_SIZE = ((arr1.getArrayLayout() == MSA_ROW_MAJOR)?
5                      arr1.getCols():arr1.getRows());
6    int numEntriesPerPage = arr1.getNumEntriesPerPage();
7    int i0 = rowStart;
8    int iFinal = rowEnd;
9    int iDelta = 1;
10   int j0 = 0;
11   int jFinal = (TheMain::COLS1 - 1);
12   int jDelta = 1;
13   int _i = i0;
14   int _j = j0;
15   do {
16     int index = ((arr1.getArrayLayout() == MSA_ROW_MAJOR)?
17                  arr1.getIndex(_i, _j):arr1.getIndex(_j, _i));
18     int indexSt = ((index / numEntriesPerPage) * numEntriesPerPage);
19     double *pi = &(arr1.getPageBottom(index, Write_Fault));
20     int indexEnd = ((indexSt + numEntriesPerPage) - 1);
21     int iSt = (indexSt / MAJOR_SIZE);
22     int iN = (indexEnd / MAJOR_SIZE);
23     int jSt = (indexSt % MAJOR_SIZE);
24     int jN = (indexEnd % MAJOR_SIZE);
25     for(;(_i <= _JADE_MIN(iFinal, iN)); _i += 1) {
26       int jEnd;
27       if ((_i == iN))
28         jEnd = _JADE_MIN(jFinal, jN);
29       else
30         jEnd = jFinal;
31       if ((_j > jFinal))
32         _j = j0;
33       for(;(_j <= jEnd); _j += jDelta) {
34         double *newname = &(pi[((((_i - iSt) * MAJOR_SIZE) + _j) - jSt)]);
35 //        int r = _i;
36 //        int c = _j;
37         {
38 //          arr1.set(r, c) = 2.0;
39           newname[0] = 1.0;
40         }
41       }
42       if ((((_i == iN) && (_j > jN)) && (_j <= jFinal)))
43         break ;
44     }
45     if ((_j > jFinal))
46       _j = j0;
47   }
48   while(((_i < iFinal) || ((_i == iFinal) && (_j <= jFinal))));
49 }
```

Figure 3.11: Jade compiler-generated code for MSA loop strip-mining.

with correspondingly efficient coherence protocols, as follows:

- *Synchronization:* Global locks were optimized by using a local proxy to minimize global communication.
- *Private:* No coherence.
- *Write-once* These are read-only after initialization. Optimized by replication.
- *Result:* Read by only one thread. Optimized by maintaining a single copy and propagating updates to it.
- *Producer-Consumer:* Optimized by eager update of copies.
- *Migratory:* Optimized by migrating the object.
- *Write-many:* Optimized by a multiple-writer protocol.
- *Read-mostly:* Optimized by replication.
- *General Read-Write:* Uses standard coherence protocol.

Their study of several shared memory programs and their performance results relative to message-passing are impressive and appear to validate their idea of "adaptive cache coherence"[18].

Munin's modes were applied on both a per-object and a per-variable basis. While TreadMarks attempts to maintain the illusion of a general shared address space, Munin requires the programmer to specify the mode for each variable. This was done at compile time and so a variable's mode could not change during the program, and only statically allocated memory was supported. Munin put each shared variable on a separate page of virtual memory.

Comparison: Munin is designed to be a complete shared memory programming model rather than the blended model of MSA. MSA supports Munin's Private and Write-many

modes, and introduces a new `accumulate` mode and prefetching commands. Munin's Write-once, Result, and Read-Mostly modes seem to be of limited use, since synchronization will be required at the application level before accessing the updated data; which leads us to believe that these modes are an artifact of Munin's static style of specifying modes. MSA accomplishes these modes by dynamically specifying a `write-many` mode followed by a `read-only` mode. Munin's Producer-Consumer mode with its eager update offers unique features, but, again, given the need for synchronization, a message send might be more efficient. MSA does not support the General Read-Write or the Read-mostly modes, in order to simplify the coherence protocol by avoiding race conditions, as explained in Section 3.1.4.

Specifying portions of an array to be in different modes is not supported in MSA, but this cannot be done in Munin either because of the static specification. Munin's granularity for data movement is the size of a virtual memory page; whereas MSA works physically on a user-defined page size. MSA's user-defined physical page size allows the "page" to be as small as one element, or as large as several rows of a matrix allowing the user (or Jade compiler) to optimize for the expected access pattern. Munin's modes are static, whereas MSA arrays can change their mode dynamically over the life of the program, which leads to needing fewer modes. Furthermore, MSA supports row-major, column-major, and (in the future) other array layouts, which can further improve performance.

DSM systems suffer considerable latency on "misses" and provide no latency tolerance mechanisms since control transfers to the DSM software in kernel space. MSA is implemented in user space, and Charm++'s virtual processors (user level threads) can tolerate latency by scheduling another virtual processor when one thread suffers a "page" miss. Section 3.5 shows performance results.

DSM uses page fault hardware to detect non-local access; MSA checks each data access (similar to Global Arrays) and we need to study the cost of this detection mechanism. MSA also has operations that work on data that is known to be available locally (e.g. using `prefetch`), and the Jade compiler can generate code for some such cases. When

combined with MSA's prefetch feature, we expect that the efficiency of array element access will approach that of sequential programs.

3.4.2 Specifically Shared Variables in Charm

Charm (and its earlier version, the Chare Kernel) supported a disciplined form of shared variables by providing abstractions for commonly used modes in which information is shared in parallel programs. [39, 36]. The modes were readonly (replicated on all processors), write-once, accumulator, monotonic variable (useful in branch-and-bound, for example), and distributed tables (basically, readonly or write-once, with distributed storage and caching). However, unlike MSA, it does not support the notion of phases, nor that of pages. Further, the original version did not have threads, and so supported a split phase interface to distributed tables.

3.4.3 Global Arrays

The Global Arrays (GA) project[50], like MSA, attempts to combine the portability and efficiency of distributed memory programming with the programmability of shared memory programming. It allows individual processes of an MIMD program to "asynchronously access logical blocks of physically distributed matrices" without requiring the application code on the other side to participate in the transfer. GA typically uses Remote DMA(RDMA)[20] and one-sided communication primitives to transfer data efficiently. GA coexists with MPI.

Each block is local to exactly one process, and each process can determine which block is local. GA provides `get`, `put`, `accumulate` (float sum-reduction), and `int read-increment` operations on individual elements of the array. The GA *synchronization* operation completes all pending transfers. *fence* waits until all transfers this process initiated are completed. Global Arrays does not implement coherence. It is the user's responsibility to guard shared access by using synchronization operations.

GA does not use page fault hardware to detect non-local access. Typically, data accesses are at the level of user-specified blocks of a matrix. Every data access is checked (with an `if`) and redirected to a local or non-local version of the operation. In GA one-sided communication is used to access a block owned by a remote processor: there is no automatic caching of remote data. The user can explicitly `fetch` remote data for extended local access and then directly access the data.

Comparison: In shared-memory terminology, it appears that GA maintains a single copy of each "page" and propagates updates to it quickly using RDMA. Unlike DSM, GA does not tie the "page" size to the VM page size but allows the user to specify an exclusive logical block of the array. It allows the block to migrate to be closer (i.e. local) to an accessing process. GA seems well-suited to certain access patterns, but, for example, implementing write-many on GA would involve a lot of unnecessary RDMA operations since there is only one home processor for a block of data, and every other processor that writes to that block does so via RDMA operations. This lack of "page" replication in GA makes reading of elements on a "page" by many threads inefficient, since, again, each read is implemented via an RDMA operation. The GA accumulate does not support variable-sized elements as MSA does. The MSA FEM example in Section 3.2.3 shows the expressivity of this feature.

3.4.4 HPF and others

Other approaches that deal with similar issues include implementation strategies for HPF[42]. For example, the inspector-executor idea [61, 41] allows one to prefetch data sections that are needed by subsequent loop iterations. HPF provides only the owner-computes model, which does not apply well to all programming problems, especially applications with dynamic irregular workloads. MSA supports owner-computes, as described in Section 3.1.4.

Obtaining full performance from MSA requires bypassing the local/remote test-condition for each access through manual fetching of blocks of data by the programmer or by using the strip-mining compiler optimization presented in Section 3.3. The simple case-based

strip-mining optimization presented only predicts monotonic access patterns. HPF (data parallel) research and other compiler technology to detect more complex access patterns[4, 2, 60] can be applied to Jade MSA programs to significantly broaden the range of access patterns detected and simplify MSA usage substantially. The user-specified MSA mode could potentially be used to simplify compiler analysis, and the MSA system can be used as a backend for data parallel code generation, providing page level data transfer instead of byte level send/receive's typically used in data parallel systems. A similar mechanism using a DSM system as the backend is presented in the paper by Min and Eigenmann[46].

Bulk Synchronous Processing[67] offers a simple parallel programming model in which all processes alternate computation with a global synchronization. Data is exchanged only during the synchronization. This model offers a simple, accurate cost model and is meant to be a bridging model connecting hardware designers and compiler writers.

Co-Array Fortran (CAF)[51] enhances Fortran95 with a very powerful notation to address remote data, and permits synchronization between all or a subset of the processes accessing a co-array. The user works with local data, and explicitly fetches and updates remote data on an as-needed basis.

ZPL[21] is a high-level parallel language for array manipulation with a powerful array notation based on working with regions of arrays. ZPL programs have a global view of both data and computation, and this is argued to improves productivity[19]. Jade MSA provides a global view of data, but computation is still specified locally in most modes. The `owner-computes` mode is an example of global specification of computation. ZPL is limited in its focus on array data, and does not provide more general data structures for dynamic, irregular applications.

Titanium is compared to Jade in Section 2.7. A recent paper[64] builds upon the inspector-executor analysis[10] to improve performance of irregularly accessed shared arrays in Titanium. It seems that the same analysis can be implemented in Jade using the MSA `prefetch` operation to prefetch the required data.

	P4 2.4 GHz		lemieux	opteron
Program	2k5k300	4096	2048	4096
Sequential C++	12.34	273	205.06	364
MSA	358			
MSA strip-mined	9	293	66.93	389

Table 3.1: Single-CPU C++ and MSA matrix multiplication execution times.

3.5 Performance Study

The computers used for our performance runs are described at the beginning of Section 2.6.

We present results for the row-wise decomposition matrix multiplication program shown in Section 3.2.1. We first compare the MSA version running on one processor to a sequential C++ matrix multiplication, in order to study the overhead introduced by MSA. The results on several machines are shown in Table 3.1. MSA without the strip-mining optimization performed by the Jade compiler is significantly slower than the sequential version. However, after applying the optimization, MSA arrays perform only slightly worse than sequential C++. An anomaly is the results on one node of the lemieux cluster, where the strip-mined MSA significantly outperforms sequential C++ matmul.

Table 3.2 below shows the scaling performance for a 2000∗5000∗300 matrix multiplication on NCSA's Tungsten cluster. When there are 8 threads per processor, the latency of page misses by a thread is better hidden by overlapping with computations for another thread. This effect (the benefit of processor virtualization) can be clearly seen by comparing results for 1 and 8 threads per processor. With a much larger number of threads per processor (32 or 64) the scheduling overhead and fine-grained communication lead to worse performance, although a more detailed study is needed to ascertain that.

The speedup plot for a larger run is shown in Figure 3.12. It should be noted that raw performance is currently unoptimized. With further optimizations, we expect the times to decrease but possibly speedups may decrease.

As mentioned earlier, the page size chosen significantly affects the performance. This can

	Processors					
Threads per processor	1	8	32	64	100	120
1	251.856	42.086	10.265	5.437	3.882	3.403
8	252.967	31.061	8.020	4.191	2.967	2.693
32	252.267	31.780	7.897	4.489	3.667	3.418
64	252.491	31.734	8.025	4.738	3.974	3.887

Table 3.2: MSA matrix multiplication (of size 2000*5000*300) execution times on Lemieux for varying number of threads per processor.

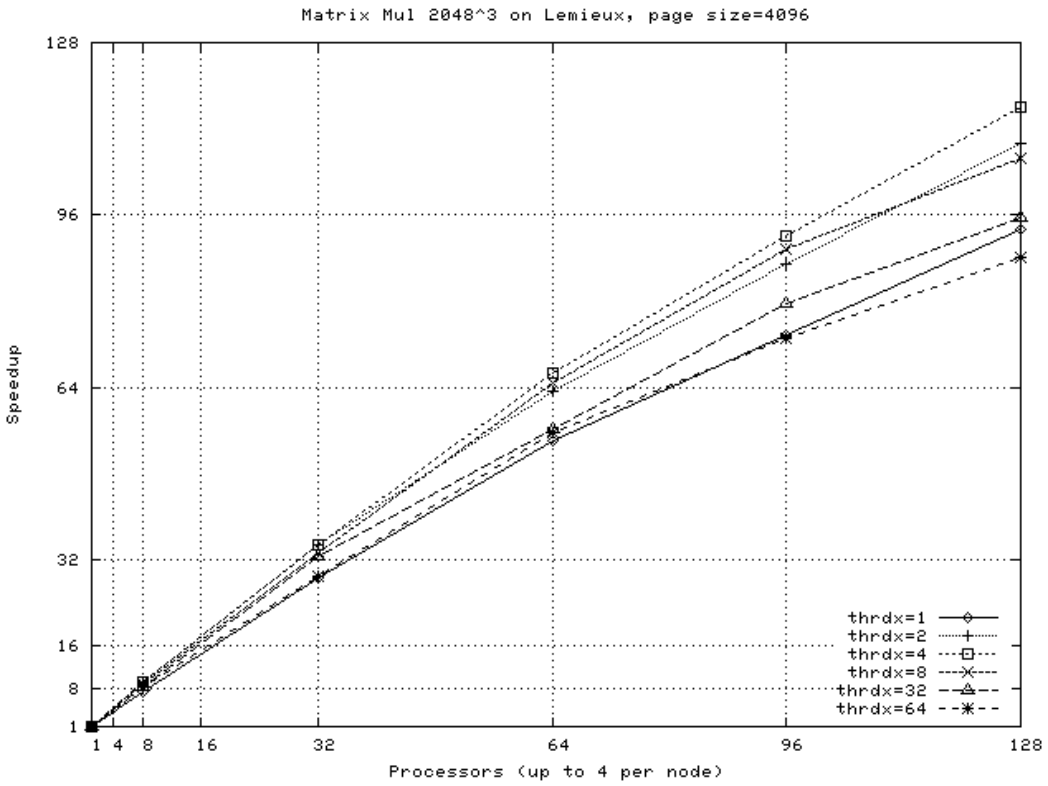

Figure 3.12: Scaling of MSA matrix multiplication (2048^3) on Lemieux with varying number of threads per processor.

be demonstrated by comparing Figure 3.12 with Figure 3.13, which shows the scaling of the same program but with a different page size. The absolute execution times on one processor for both cases are almost the same (about $550s$), allowing us to compare the speedups of the two cases. The maximum speedup on 128 processors drops to about 80 for the page size of 1024 elements as compared to about 116 for the previous case.

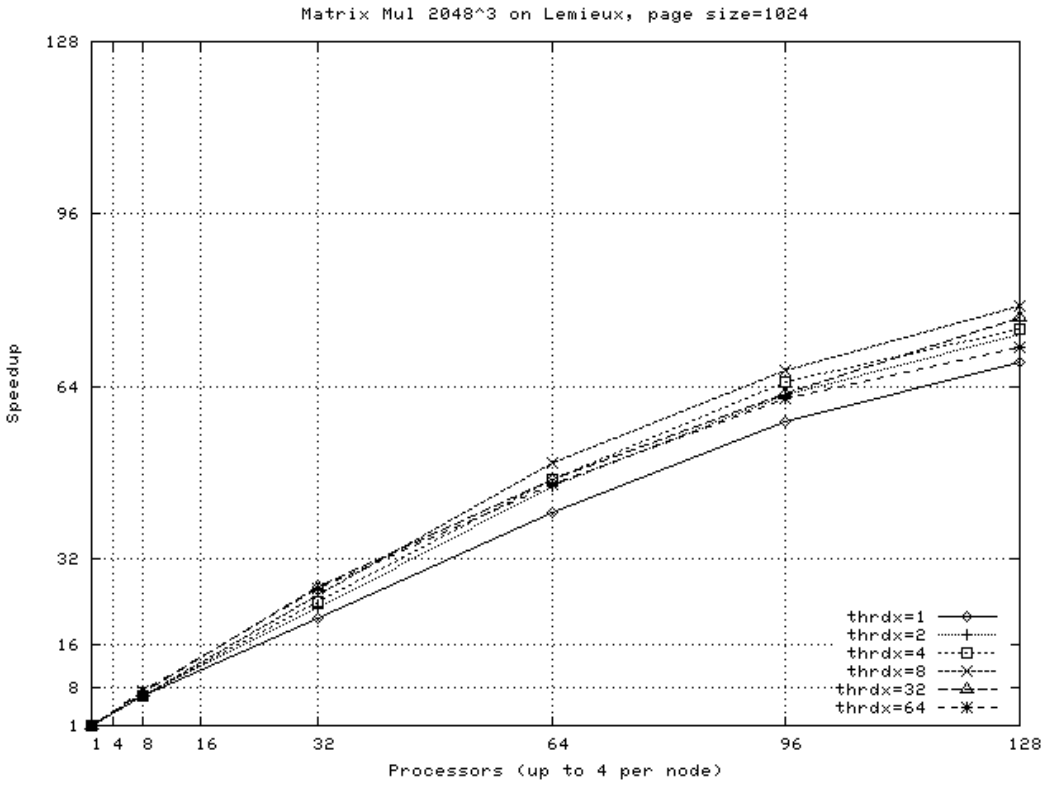

Figure 3.13: Scaling of MSA matrix multiplication (2048^3) on Lemieux using page size of 1024.

Figure 3.14 shows the effect of limiting the MSA local processor page replication cache size: with a smaller cache, the time is almost twice as large as that with an adequately large cache. Smaller caches reduce the reuse of fetched data. Although our performance studies have assumed a sufficiently large cache, MSA's offer an elegant solution to situations where the application data is too large to be replicated into local memory on every processor. This happens for the FEM example described in Section 3.2.3. Studies of the cache performance for various page replacement policies and applications are an avenue for further exploration.

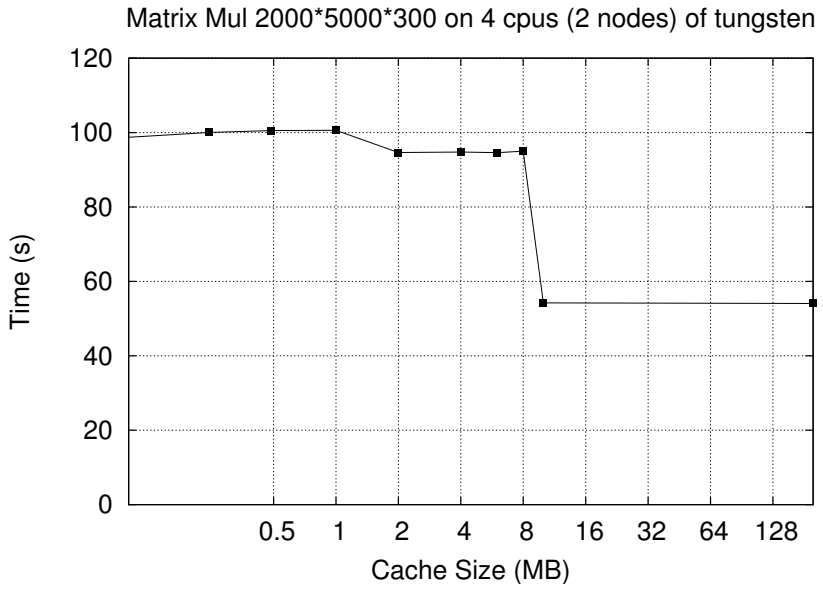

Figure 3.14: Effect of varying MSA cache size for MSA matrix multiplication (2000*5000*300) on Tungsten.

Finally, the performance results of the MSA molecular dynamics example described in Section 3.2.2 are shown in Figure 3.15. The application computes VDW and electrostatics forces for all atom interactions and uses cutoff distances to limit computation, but ignores the integration step that moves atoms. To compare MSA with Charm++, a similar Charm++ program was implemented and the results are also shown in the graph. The Charm++ program uses the same 2D decomposition of the iteration space and handles communication by communicating chunks of the atom array to the worker threads that need the corresponding chunks. After the worker threads complete a timestep computation, the results are sent back to the atom array, which sums them up. The execution times of both the Charm++ and MSA programs are shown in Table 3.3.

3.5.1 Responsiveness Issues in MSA

Table 3.4 shows the execution times on Lemieux for a matrix multiplication program whose cache performance was optimized by interchanging the two inner loops and switching the B matrix to row major data layout. Performance did not scale as expected. We can use

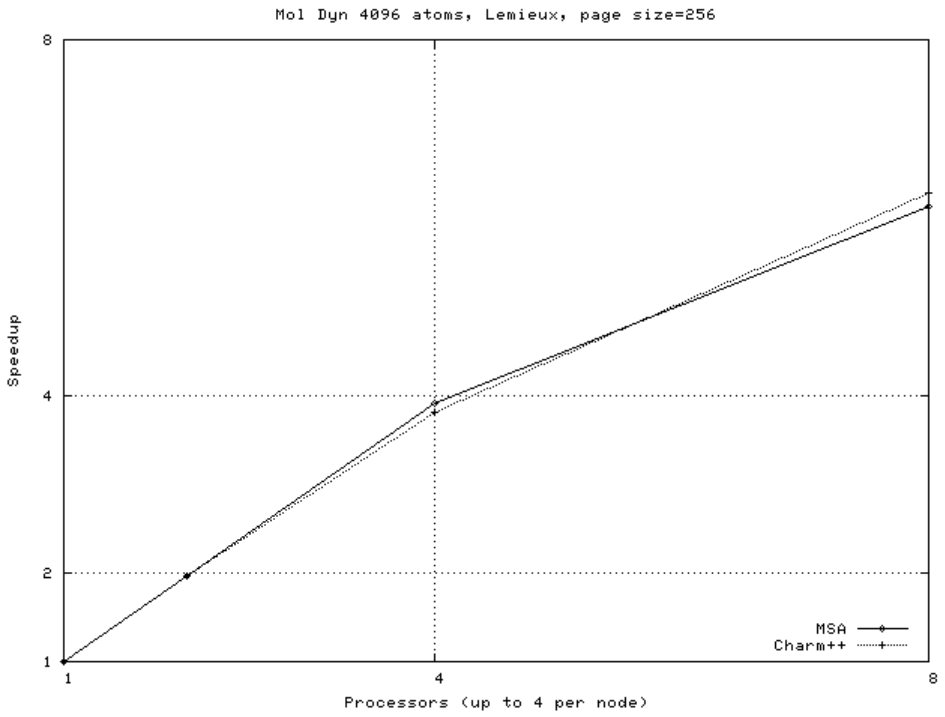

Figure 3.15: Scaling of the Plimpton-style molecular dynamics example program on LeMieux: MSA and plain Charm++.

CPU's	MSA	Charm++
1	20.89	18.75
2	10.63	9.53
4	5.35	4.93
8	3.41	2.98

Table 3.3: MSA and Charm++ molecular dynamics execution times on Lemieux. 4096 atoms with 2D 16x16 decomposition (256 threads) and page size=256.

Threads per processor	Processors		
	1	2	4
1x	19.24287	11.668164	20.198651
4x	20.902221	21.752528	24.398671

Table 3.4: Optimized MSA matrix multiplication (of size 2000*5000*300) execution times on Lemieux for varying number of threads per processor.

the Projections performance visualization tool to study the program behavior. Figure 3.16 shows the Projections Overview Graph for a 4-cpu/16-worker run of the program. The x-axis represents time; the y-axis is divided into equal bands for each processor and the color of the interior depicts the processor utilization. White represents maximum load and black 0% utilization. Examination of the overview graph reveals that the processors are delayed significantly. Average utilization of processors, shown in the Projections Usage Profile graph in Figure 3.17, is less than 50%. Analyzing the detailed performance using Projections Timeline Tool (Figure 3.18) shows that Processor 3 is delayed waiting for page data to arrive from Processor 2.

The cause of the delay is that page requests have the same priority as any other Charm++ messages. Since the worker messages are enqueued in the processor message queue before the page requests arrive, they are handled first, delaying other processors that require the data to proceed. To test our hypothesis, we inserted a `CthYield` call in our matrix multiplication loop that would yield after computing 500 elements. Figure 3.5 shows the improved execution times. Figures 3.19 and 3.20 show the Projections overview graph and usage profile respectively.

The design solution to this problem is to permit page requests to have a high priority; even better, page requests should interrupt an executing thread and be immediately handled for best performance.

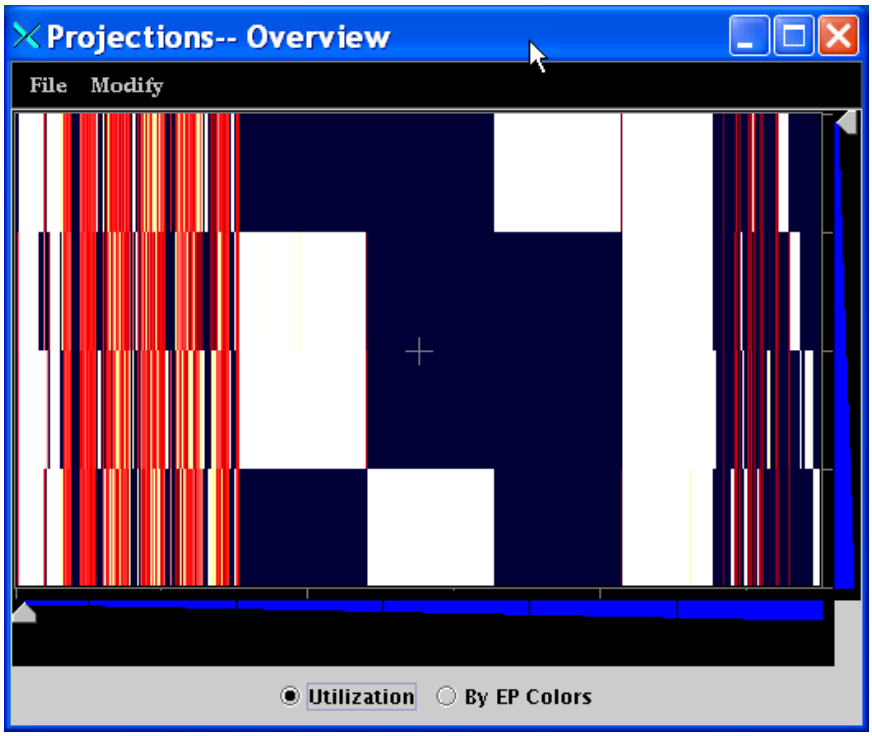

Figure 3.16: Projections performance overview graph for 4-cpu matrix multiplication on Lemieux.

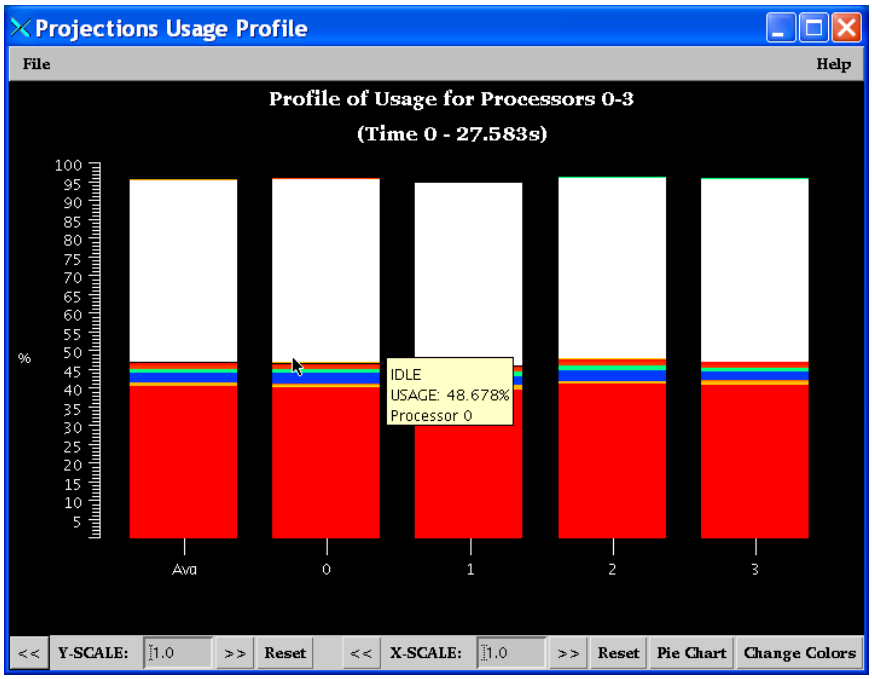

Figure 3.17: Projections usage profile for 4-cpu matrix multiplication on Lemieux.

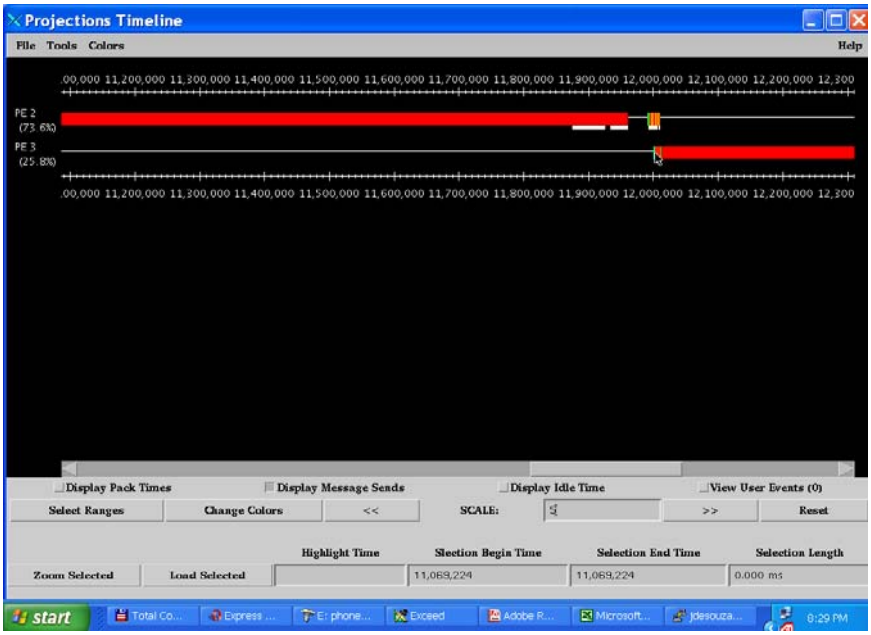

Figure 3.18: Projections timeline graph for 4-cpu matrix multiplication on Lemieux.

Threads per processor	Processors		
	1	2	4
1x	19.683789	12.033342	8.742496

Table 3.5: Optimized MSA matrix multiplication (of size 2000*5000*300) execution times on Lemieux after using `CthYield`.

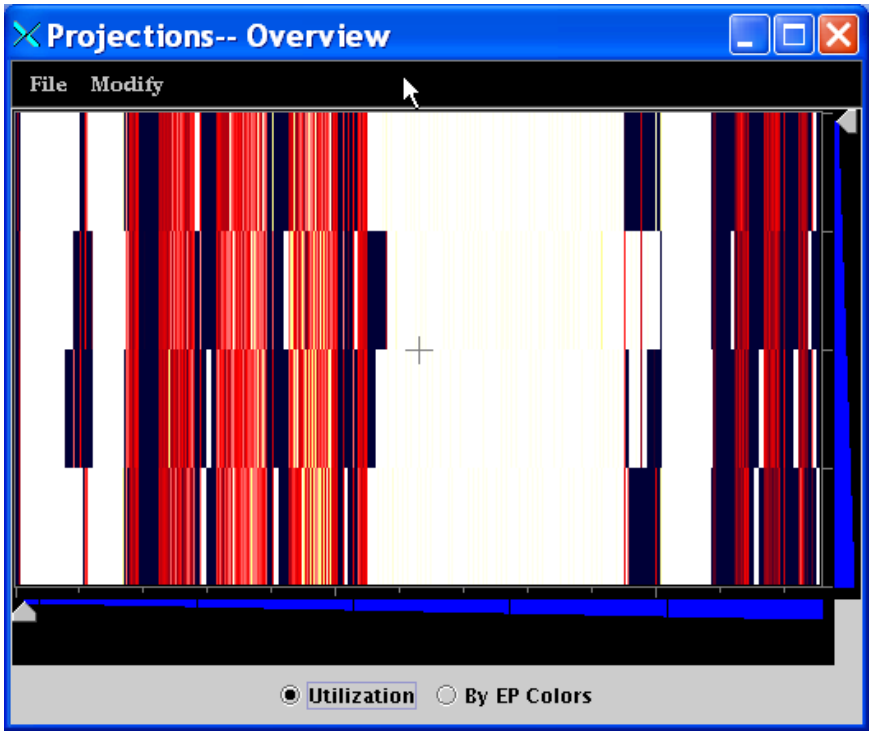

Figure 3.19: Projections performance overview graph for 4-cpu matrix multiplication on Lemieux after using `CthYield`.

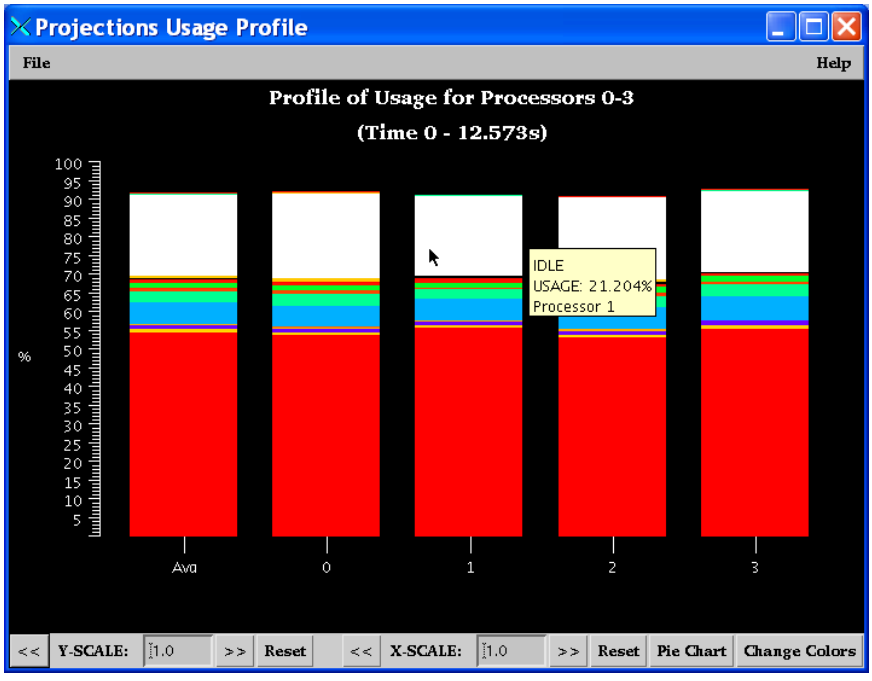

Figure 3.20: Projections usage profile for 4-cpu matrix multiplication on Lemieux after using `CthYield`.

3.6 Summary

We described a restricted shared address space (SAS) programming model called multi-phase shared arrays (MSA), its implementation and its use with examples. MSA is not a complete model, and is intended to be used in conjunction with other information-exchange paradigms such as message passing. It currently supports four modes: `read-only`, `write-many`, `accumulate`, and `owner-computes`. An important feature of MSA arrays is that the modes for each array can change dynamically in different phases of the program separated by synchronization points. User-selectable page sizes offer scope for maximizing performance, in contrast to the fixed virtual memory page sizes of DSM approaches. Variable-sized data elements and the generalized `accumulate` operation supported by MSA are powerful and can be used to accumulate by set union, list append, etc. in addition to the more common use in summations. MSA is implemented in Jade, Charm++ and AMPI, which support many light-weight threads (virtual processors) per processor. As a result, the latency inherent in page misses is better tolerated. Further, we provide a page fetch operation and correspondingly specialized versions of array accesses, that attain the efficiency of sequential code when we apply the Jade strip-mining compiler optimization to fetch remote pages and then access the data locally.

We hope that the mixed mode model provided by MSA and Jade will lead to substantial improvement in programmer productivity, and bridge the current divide between SAS and distributed memory programming styles. We have identified several avenues for further study in the final chapter of this thesis.

Chapter 4

Compiler-supported Live Data Migration

The processor virtualization approach insulates the program from the number of processors, and uses the adaptive runtime system(ARTS) to manage work objects and implement automatic measurement-based load-balancing. The ARTS responds to load imbalance by redistributing work through migration of work objects. Object migration is the cornerstone of the load balancing implementation in Jade, Charm++ and AMPI.

We use compiler support in Jade to reduce the amount of data migrated and the time taken to migrate objects by the simple expedient of packing and unpacking only data that is live when the object migrates. A Jade parallel object can contain multiple threads of execution. At any given time, these threads may have reached different points in their execution path. Therefore, the set of variables that are live depends on the execution state of each sub-thread within the parallel object. A mechanism is needed to handle the combinatorial cases that can occur when migrating an object.

Our user-assisted compiler-supported approach, described in Section 4.1, requires the user to declare variables live and dead. (The `live` keyword is needed in order to make a variable live at the start of a new phase or iteration within the program.) The Jade compiler generates code to track the set of live/dead variables, and pack/unpack(`pup`) only live data.

Another benefit that accrues from the enhancement of the Jade `pup` mechanism, is that

checkpointing also enjoys the same benefits as migration. This is due to the fact that in Jade/Charm++, checkpointing is implemented by packing an object into a "message" that is written to disk.

A full compiler approach that performs live variable analysis in Jade to automatically determine the liveness of data without programmer assistance would be a natural next step. In addition to automating this feature, such analysis can determine optimal migration points by identifying the points of the program at which the amount of live data is least. Multi-threaded Jade parallel objects complicate liveness analysis due to the combinatorial states the object can be in based on the execution point of each individual sub-thread of the object. Another source of complication is the message-driven nature of Jade: the lack of a clear thread of control expressed in the code makes live variable analysis difficult. We discuss the issues in more detail and propose a design to resolve them in Section 4.2.

Performance of our user-assisted approach is very promising and is discussed in Section 4.3. We conclude the chapter by listing some possibilities for further study.

4.1 User-assisted Compiler-supported Object Migration and Checkpointing

The basic pack-unpack mechanism available in Jade packs up all the data members of an object. This mechanism can be optimized for size by packing only the live data members.

As mentioned earlier, the several threads of a multi-threaded parallel Jade object can each be at a different point in their execution when it is time to migrate/checkpoint the parallel object. Therefore, code cannot be generated at various migration points to pack a fixed set of data, since the data that is live depends on the state of the individual threads within the object.

A mechanism is needed that keeps track of the state of each variable as the various threads within the object execute. Then, only live data should be migrated at a migration

point.

As described in the following section, liveness analysis within Jade introduces some complexities that we have left for further study. We have implemented a user-assisted approach, in which the programmer is required to indicate when a class variable becomes dead.

We use compiler support, as described below, to keep track of the variables within the parallel object, and added the `dead` and `live` keywords to the Jade language to specify the state of individual variables. Alternative solutions require substantial overhead for the programmer, as we discuss after describing the solution we chose.

Thus `dead A;` declares the object A as dead. `dead` may be applied to primitive types and sequential arrays or objects in Jade. (For a discussion of the kinds of data in Jade, the reader is referred to Section 2.2.1.)

`dead` may also be applied to parallel entities in Jade, but in Jade, variables declared as parallel entities are actually proxies that contain a reference to the actual entity, and declaring them as dead only declares the proxy as dead, not the parallel entity itself. These proxies are very tiny objects and in most cases it may not be worthwhile to declare them as dead.

The Jade compiler performs the following steps to implement the pupping of live variables:

- The compiler maintains a list of all the class variables in each `class` declared.
- It generates code that declares a bit-vector of the appropriate size in each class. This bit-vector is used to keep track of which class variables are dead.
- The compiler converts use of the `dead` keyword into code that sets the appropriate bit in the deadness bit-vector. The `live` keyword clears the bit.
- When generating the pack/unpack code, the compiler packs/unpacks the bit-vector first followed by only the live variables. When unpacking, only the variables whose bits are clear in the bit-vector are unpacked into the class member data.

One point of interest that should be mentioned, is that a variable declared as `dead` cannot necessarily be garbage collected, since it may be declared `live` in the future. Stack-allocated data also cannot be garbage collected, even if declared `dead`. Thus, it should be clarified that the Jade semantics of `dead` do not automatically deallocate the data.

Alternatives to the solution proposed would essentially require the programmer to perform the above steps. There is no clean way to keep track of class variables using a library API; some kind of registration of each class variable would be required. An alternative to defining the `dead` and `live` keywords would be define a global `dead` method and an enum within each class to refer to each class variable, or to require a programmer-defined or compiler-generated `dead` method within every sequential class. The latter solution would not work for primitive types. None of these alternatives is as clean as our compiler-supported approach.

The performance of this approach is presented in Section 4.3. In the next section, we discuss the issues with performing live variable analysis in Jade.

4.2 Design for Live Variable Analysis in Jade

An issue arises when analyzing pure message-driven (MD) programs for live variables. As mentioned earlier, the message-driven nature of Charm++ requires a single conceptual procedure to be split into multiple non-blocking functions, thus obscuring the control flow within an object. For example, at line 24 in Figure 2.1 the `SayHi` method is invoked asynchronously, i.e. the code does not block and wait for control to return. Execution of the `Main` method continues until it is complete. In order for execution to resume after `SayHi` conceptually "returns", we need to introduce the `done` method (line 27) which is called back from line 53 after keeping track of the number of elements executed.

Now, back to the problem: In a pure message-driven program, there would be no way to determine the liveness of an object's member variables since the control flow logic is buried in

the logic of the program code. We cannot always determine whether a method of an object has been invoked for the last time. As a corollary, we would be unable to determine when to garbage collect the object since there is no thread of control for the object that explicitly terminates.

One solution would be to make use of threads. By forcing objects to be threaded with a single thread of control, standard liveness analysis can be performed, essentially treating the threaded method as `main`. Although threads permit object control flow to be expressed within a single method, they must be used sparingly to retain the message-driven benefits of latency tolerance, synchronization-freedom, and easy migration.

As an alternative to threads, a solution based on the existing Structured Dagger (SDAG) language can be used. SDAG is a preprocessor language used to express the life cycle of a Charm++/Jade object. SDAG is translated into pure message-driven Charm++ which does not use any threads. Consider the following example SDAG code from [13]:

```
1  class compute_object
2  sdagentry compute_object(MSG *msg){
3  atomic{
4    P->get(msg->first_index, ...);
5    P->get(msg->second_index, ...);
6  }
7  overlap{
8    when recv_first(Patch *first) atomic {filter(first);}
9    when recv_second(Patch *second) atomic {filter(second);}
10 }
11 atomic { computeInteractions(first, second);}
12 }
```

`compute-object` fires off two requests for data in the first `atomic` block of code. The responses might arrive in either order, which is handled by the `overlap` and `when` statements. After both responses are received and processed, execution continues at the final atomic block. The use of SDAG clearly expresses the life-cycle of a message-driven (MD) object. Using this information, live variable analysis (LVA) can be performed.

Returning to liveness analysis, we have the following cases:

	Checkpoint Size (KB)		Checkpoint Time	
	1 cpu	2 cpus	1 cpu	2 cpus
NFS Disk				
without optimization	131740	131760	12.780526	12.37902
with optimization	65884	65904	6.178298	6.113425
Local Disk				
without optimization	131729	131740	10.781935	2.871472
with optimization	65872	65884	3.307590	1.561786

Jacobi 2D array with 2D decomposition, data array 4096*4096 floats, decomposition 4*4 = 16 `ChareArray` elements. Executed on two P4 2.4 GHz single-processor nodes connected by 100 Mbps Ethernet.

Table 4.1: Checkpoint size and time for pup-optimized Jacobi with 2D data matrix of size 4096*4096 elements and 2D decomposition into 4*4 sections.

- Pure MD, no SDAG: We cannot do LVA for object variables.
- Pure MD, SDAG object: We can do LVA.
- One main thread, no public `entry methods` (AMPI-like): We can do LVA.
- Other cases: We can do LVA within thread, but not for object

Therefore, the proposed solution is to incorporate an SDAG-like facility into Jade, and enforce the following rule in the language: an object must either use SDAG, or it may have only one threaded method and no public `entry methods`. This restriction is not as limiting as it seems, since it effectively permits Charm++ message-driven and AMPI message-passing programming, respectively.

4.3 Performance Results

Table 4.1 shows the performance of checkpointing a Jacobi program, in terms of data size written to disk and time taken for the checkpointing. We executed the program with all processors writing to a single NFS-mounted directory, and we also tested the case of processors writing to local disk.

In both tests, as expected, the data size with the optimized pup is about half the regular pup, resulting in significant space savings.

Looking at the NFS-based test results, we see that the time taken to complete the checkpointing does not decrease as expected for the 2-cpu case over the 1-cpu case. However, for the local-disk version the performance of the 2-cpu case is much better than the 1-cpu case. This leads us to believe that the NSF performance delays the processors.

In absolute terms, the local-disk tests are much faster than writing via NFS. This behavior is expected; however, local disk checkpointing cannot always be used since the local disk will go down along with a node if a node fails, making the checkpoint useless. Of course, for application failure (or other faults such as time-limit expiration) local disks may still be useful. A solution to this issue is the in-memory (which subsumes local disk) double checkpointing described in [71]. Our pup optimizations can make use of the mechanism described there without any changes.

4.4 Conclusion

We described a user-assisted compiler-supported mechanism in Jade that improves the performance of object migration, checkpointing, and parameter passing. The mechanism requires the user to specify when variables are no longer live, and uses the compiler to ensure that only live variables are packed when the data needs to be transferred. Performance results are very promising.

We also discussed the issues involved in performing live variable analysis for message-driven programs, and proposed a design solution to address these issues. This needs to be studied further and implemented.

Currently, data is optimized irrespective of its size; this involves keeping track of its status, and performing a check when the data is packed or unpacked. A more efficient solution is to completely avoid checking of small data structures and only generate code to

check `dead` for large structures.

Chapter 5

Conclusion and Future Work

The main focus of this thesis is to enhance productivity in the programming of processor-virtualization based (PV) systems.

We identified several issues with current-generation PV languages that interfere with programmer productivity. Many of the issues identified are syntactic: some of these are the necessity of writing a translator file, the lack of parameter marshalling in remote method invocation, and the tedious pack/unpack overheads. Other issues are design issues, such as the complexity of user memory management, the aliasing issues that accompany general pointer support, and the usefulness of HPF-style array section notation. We argued for a compiler based solution and a newly designed programming language, and designed and implemented the Jade programming language and compiler framework. We implemented several applications in Jade and studied and compared the productivity benefits and performance of Jade with Charm++.

We identified a key limitation in the multi-paradigm support of current PV languages: the lack of a general and efficient shared-address-space programming model. We presented a solution, the multiphase specifically shared array (MSA) programming model, described its design and implementation in Jade and as a Charm++ library, wrote several applications using the model, and studied its expressivity and performance. The performance of the MSA model was optimized using a strip-mining pass in the Jade compiler that improved MSA performance significantly, putting it on par with locally accessed sequential data arrays.

The Jade multi-pass compiler framework also provides a foundation for further PV language research, including compiler optimizations. We identified one such optimization, reducing checkpoint size and time in multi-threaded objects, we designed a user-assisted solution, and implemented it in the Jade compiler. We compared the simplicity (from the programmer's viewpoint) of this compiler-supported solution with the complexity of a non-compiler-supported solution, presenting it as further evidence of the benefits of compiler support.

We believe that Jade and MSA substantially simplify parallel programming of processor-virtualization based systems while retaining the associated performance benefits.

5.1 Future Work

The implementation of the Jade language is not complete. The implementation of some features has been deferred, although design solutions for their implementation have been studied. These include:

- garbage collection of sequential and parallel objects
- packing of multiply-referenced data structures
- live-variable analysis of the modified design described in Section 4.2.
- Migration of multi-threaded objects, discussed in Section 2.1.3.

Although the focus of Jade is to improve on PV languages, a detailed comparison of the expressivity and performance of Jade with programming languages such as Titanium, Co-array Fortran, and ZPL are an avenue for study. Some of their notational conveniences and compiler analyses could be added to Jade.

Another compiler optimization could take advantage of MSA `prefetch` support. Jade can insert `prefetch` calls and generate two versions of a section of user code: one for known-local access of all shared variables, the other for the unknown-if-local case. At the beginning

of a phase a check is performed to determine whether the shared variables accessed are all available locally. If so, the known-local case is executed. Otherwise, when possible, we issue prefetches and then execute the known-local case. If it is not possible to determine the access we execute the unknown-if-local case.

More MSA applications need to be identified and written. A couple are being implemented by other researchers at the Parallel Programming lab. Additional access modes beyond `read-only`, `write-many`, `accumulate` and `owner-computes` can be explored: one possibility is a producer-consumer mode. Further performance optimization, and detailed performance studies, including studies of various page-replacement policies for MSA-cache-limited applications can be done. Finally, compiler support is crucial to MSA performance and simplifying use of MSA, which can be explored in the Jade programming language and compiler framework: for example compiler-generated prefetch calls.

Appendix A

Jade Example Programs

A.1 Jade non-MSA Matrix Multiplication

```
package JadeMatmul;

public synchronized class TheMain extends Chare {
    public static readonly CProxy_TheMain mainChare;
    public static readonly int M, N, K;  // matrix problem size
    public static readonly int NUM_CHUNKS; // decomposition

    public int main(String []args){
        mainChare = thisProxy;
        M = 2000;
        K = 5000;
        N = 300;
        NUM_CHUNKS = CkNumPes(); // As many workers as processors

        ChareArray1D mm = new Matmul[NUM_CHUNKS];
        mm.mul(); // Broadcast
    };

    public void mainDone() { CkExit(); };
};

public synchronized class Matmul extends ChareArray1D {
    double[][] A, B, C;
    int iStart, iEnd;

    public Matmul() {
        int rangeSize = TheMain.M / TheMain.NUM_CHUNKS;
        iStart=CkMyPe()*rangeSize;
        iEnd=iStart+rangeSize;

        A = new float[TheMain.M][TheMain.K];  // create the whole A, use only our part.
        B = new float[TheMain.K][TheMain.N];
```

```
        C = new float[TheMain.M][TheMain.N];
        init();
    }

    public void mul() {
        for (int i=iStart; i<iEnd; i++)
            for (int k=0; k<TheMain.K; k++)
                for (int j=0; j<TheMain.N; j++) {
                    C[i][j] += A[i][k] * B[k][j];
                }

        // Reduction calls mainDone()
        CkCallback cb = new CkCallback(TheMain.mainDone, TheMain.mainChare);
        contribute(1, thisIndex, CkReduction.sum_int, cb);
    }

    private void init() {
        for (int i=iStart; i<iEnd; i++)
            for (int k=0; k<TheMain.K; k++)
                A[i][k] = 1.0;

        for (int k=0; k<TheMain.K; k++)
            for (int j=0; j<TheMain.N; j++)
                B[k][j] = 1.0;
    }

}
```

A.2 Jacobi

```
package Jacobi2D;

public synchronized class TheMain extends Chare {
    public static readonly CProxy_TheMain mainChare;
    // Total size of each dimension of data array = CHUNK_SIZE * NUM_CHUNKS
    public static final int NUM_CHUNKS=10; // e.g. 100*100 chunks
    public static final int CHUNK_SIZE=16; // e.g. 64*64 elements per chunk
    public static final int NUM_ITERATIONS=100;

    public int main(String []args){
        mainChare = thisProxy;

        ChareArray2D jc = new JacobiChunk[NUM_CHUNKS][NUM_CHUNKS];
        jc.start(NUM_ITERATIONS);          // Broadcast
    };

    public void mainDone() { CkExit(); };
};

public synchronized class JacobiChunk extends ChareArray2D {
```

```
float[][] data;  // = new float[1+CHUNK_SIZE+1][1+CHUNK_SIZE+1];
int numIters;  // How many iterations left.
int numGot;    // in each iteration: have I received data from all
               // neighbors ?
int numDone;   // termination: how many ChareArray elements are
               // done with their iterations?
int numNeighbors;

// Constructor, initializes data
public JacobiChunk(){
    int i, j;
    data = new float[1+TheMain.CHUNK_SIZE+1][1+TheMain.CHUNK_SIZE+1];
    // Fill data with 101, 99, 101, 99, 101, ...
    for(i=1; i<=TheMain.CHUNK_SIZE; i++)
        for(j=1; j<=TheMain.CHUNK_SIZE; j++)
            data[i][j] = 100.0+ ((i+j)%2?-1:1);

    numGot = 0;
    numIters = 0;
    numDone = 0;
    numNeighbors = 4;
    if (thisIndex.x==0 || thisIndex.x==TheMain.NUM_CHUNKS-1)
        --numNeighbors;
    if (thisIndex.y==0 || thisIndex.y==TheMain.NUM_CHUNKS-1)
        --numNeighbors;
    maxDelta = 0.0;
};

public void start(int nIters) {
    numIters = nIters;
    startNextIter();
};

// push data to consumers
public void startNextIter() {
    if (numIters > 0) {
        int i;
        numIters--;
        // send Up
        if (thisIndex.x > 0)
            thisProxy[thisIndex.x-1][thisIndex.y].getBottom(
              data[1][0:TheMain.CHUNK_SIZE+1]);

        // send Down
        if (thisIndex.x < (TheMain.NUM_CHUNKS-1))
            thisProxy[thisIndex.x+1][thisIndex.y].getTop(
              data[TheMain.CHUNK_SIZE][0:TheMain.CHUNK_SIZE+1]);

        // send Left
        if (thisIndex.y > 0)
            thisProxy[thisIndex.x][thisIndex.y-1].getRight(
              data[0:TheMain.CHUNK_SIZE+1][1]);
```

```
        // send Right
        if (thisIndex.y < (TheMain.NUM_CHUNKS-1))
            thisProxy[thisIndex.x][thisIndex.y+1].getLeft(
              data[0:TheMain.CHUNK_SIZE+1][TheMain.CHUNK_SIZE]);

    } else { // this array element's iterations are finished
        // inform 0 that we are done
        thisProxy[0][0].done(0);
    }
};

public void getLeft(float[][] left) {
    for(int i=1; i<=TheMain.CHUNK_SIZE; i++)
        data[i][0] = left[i][0];
    compute();
};

public void getRight(float[][] right) {
    for(int i=1; i<=TheMain.CHUNK_SIZE; i++)
        data[i][TheMain.CHUNK_SIZE+1] = right[i][0];
    compute();
};

public void getTop(float[][] top) {
    for(int i=1; i<=TheMain.CHUNK_SIZE; i++)
        data[0][i] = top[0][i];
    compute();
};

public void getBottom(float[][] bottom) {
    for(int i=1; i<=TheMain.CHUNK_SIZE; i++)
        data[TheMain.CHUNK_SIZE+1][i] = bottom[0][i];
    compute();
};

public void compute() {
    if (++numGot == numNeighbors)
        numGot = 0;
    else // not yet received all neighbors
      return;

    // compute
    for (i=1; i<=TheMain.CHUNK_SIZE; i++)
        for (j=1; j<=TheMain.CHUNK_SIZE; j++)
            data[i][j] = (data[i-1][j] + data[i][j] +
              data[i+1][j] + data[i][j-1] + data[i][j+1])/5.0;

    // Do a reduction for synchronization
    CkCallback cb = new CkCallback(JacobiChunk.callBackTarget,
                                   thisProxy[0][0]);
    contribute(1, thisIndex.x, CkReduction.sum_int, cb);
};
```

```
    public void callBackTarget(int i) {
        // Go to next iteration
        thisProxy.startNextIter();  // broadcast
    }

    public void done(float delta) {
        ++numDone;
        if (numDone == TheMain.NUM_CHUNKS*TheMain.NUM_CHUNKS) {
            TheMain.mainChare.mainDone();
        }
    }
}
```

Appendix B

MSA Examples in Charm++ Notation

B.1 FEM Example

```
1  // Phase I: EtoN: RO, NtoE: Accu
2  for i=1 to EtoN.length()
3    for j=1 to EtoN[i].length()
4      n = EtoN[i][j];
5      NtoE[n].accumulate(i);  // set accumulate operation
6
7  // Phase II: NtoE: RO, EtoE: Accu
8  for j = my section of NtoE[j]
9    //foreach pair e1, e2 elementof NtoE[j]
10   for i1 = 1 to NtoE[j].length()
11     for i2 = i1 + 1 to NtoE[j].length()
12       e1 = NtoE[j][i1];
13       e2 = NtoE[j][i2];
14       EtoE[e1].accumulate(e2); // set accumulate
15       EtoE[e2].accumulate(e1); // set accumulate
```

Figure B.1: Charm++ version: MSA FEM Code.

```
1  typedef MSA2D<double, MSA_NullA<double>, 5000, MSA_ROW_MAJOR> MSA2DRowMjr;
2  typedef MSA2D<double, MSA_SumA<double>, 5000, MSA_COL_MAJOR> MSA2DColMjr;
3
4  // One thread/process creates and broadcasts the MSA's
5  MSA2DRowMjr arr1(ROWS1, COLS1, NUMWORKERS, cacheSize1); // row major
6  MSA2DColMjr arr2(ROWS2, COLS2, NUMWORKERS, cacheSize2); // column major
7  MSA2DRowMjr prod(ROWS1, COLS2, NUMWORKERS, cacheSize3); // product matrix
8
9  // broadcast the above array handles to the worker threads.
10 ...
11
12 // Each thread does the following code
13 arr1.enroll(numWorkers); // barrier
14 arr2.enroll(numWorkers); // barrier
15 prod.enroll(numWorkers); // barrier
16
17 while(iterate)
18 {
19     for(unsigned int c = 0; c < COLS2; c++) {
20         // Each thread computes a subset of rows of product matrix
21         for(unsigned int r = rowStart; r <= rowEnd; r++) {
22             double result = 0.0;
23             for(unsigned int k = 0; k < cols1; k++) {
24                 double e1 = arr1.get(r,k);
25                 double e2 = arr2.get(k,c);
26                 result += e1 * e2;
27             }
28
29             prod.set(r,c) = result;
30         }
31     }
32
33     prod.sync();
34     // use product matrix here
35 }
```

Figure B.2: Charm++ version: MSA Matrix Multiplication Code.

B.2 Matrix Multiplication

B.3 Molecular Dynamics

```
1  // Declarations of the 3 arrays
2  class XYZ; // { double  x; double y; double z; }
3  typedef MSA1D<XYZ, MSA_SumA<XYZ>, DEFAULT_PAGE_SIZE> XyzMSA;
4  class AtomInfo;
5  typedef MSA1D<AtomInfo, MSA_SumA<AtomInfo>, DEFAULT_PAGE_SIZE> AtomInfoMSA;
6  typedef MSA2D<int,MSA_NullA<int>,DEFAULT_PAGE_SIZE,MSA_ROW_MAJOR> NeighborMSA;
7
8  XyzMSA coords;
9  XyzMSA forces;
10 AtomInfoMSA atominfo;
11 NeighborMSA nbrList;
12
13 //broadcast the above array handles to the worker threads.
14 ...
15
16 // Each thread does the following code
17 coords.enroll(numberOfWorkerThreads);
18 forces.enroll(numberOfWorkerThreads);
19 atominfo.enroll(numberOfWorkerThreads);
20 nbrList.enroll(numberOfWorkerThreads);
21
22 for  timestep = 0 to Tmax {
23   /*************** Phase I ***************/
24   // for a section of the interaction matrix
25   for i = i_start to i_end
26     for j = j_start to j_end
27       if (nbrList.get(i,j)) { // nbrList enters ReadOnly mode
28         force = calculateForce(coords[i], atominfo[i], coords[j], atominfo[j]);
29         forces.accumulate(i, force); // Accumulate mode
30         forces.accumulate(j, -force);
31       }
32   forces.sync();
33
34   /*************** Phase II ***************/
35   for k = myAtomsbegin to myAtomsEnd
36      coords.set(k, integrate(atominfo[k], forces[k])); // WriteOnly mode
37   coords.sync();
38
39   /*************** Phase III ***************/
40   if  (timestep %8 == 0) { // update neighbor list every 8 steps
41     // update nbrList with a loop similar to the force calculation loop above
42     ... nbrList.set(i, j, value);
43
44     nbrList.sync();
45   }
46 }
```

Figure B.3: Charm++ version: MSA Molecular Dynamics Example Code.

References

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[2] Vikram Adve and John Mellor-Crummey. Using integer sets for data-parallel program analysis and optimization. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 186–198. ACM Press, 1998.

[3] V.S. Adve, J. Mellor-Crummey, M. Anderson, K. Kennedy, J.-C. Wang, and D.A. Reed. An Integrated Compilation and Performance Analysis Environment for Data Parallel Programs. In *Proceedings of Supercomputing'95*, December 1995.

[4] Saman P. Amarasinghe and Monica S. Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 126–138. ACM Press, 1993.

[5] Darius Antia and Franck van Breugel. Semantic analysis of pict in Java. Technical Report CS-2003-10, Department of Computer Science, York University, Toronto, Ontario M3J 1P3, Canada, October 2003.

[6] Pedro V. Artigas, Manish Gupta, Samuel P. Midkiff, and Jose E. Moreira. High performance numerical computing in Java: Language and compiler issues. In *Languages and Compilers for Parallel Computing*, pages 1–17, 1999.

[7] Henri Bal et al. Manta: Fast parallel Java. URL: http://www.cs.vu.nl/manta/.

[8] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proc. of the Second ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'90)*, pages 168–177, 1990.

[9] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Adaptive software cache management for distributed shared memory architectures. In Igor Tartalja and Veljko Milutinovic, editors, *The cache coherence problem in shared memory multiprocessors: software solutions*. IEEE Computer Society Press, 1995.

[10] Harry Berryman and Joel Saltz. *A manual for PARTI runtime primitives*, 1990.

[11] Milind Bhandarkar and L. V. Kalé. A Parallel Framework for Explicit FEM. In M. Valero, V. K. Prasanna, and S. Vajpeyam, editors, *Proceedings of the International Conference on High Performance Computing (HiPC 2000), Lecture Notes in Computer Science*, volume 1970, pages 385–395. Springer Verlag, December 2000.

[12] Milind Bhandarkar, L. V. Kale, Eric de Sturler, and Jay Hoeflinger. Object-Based Adaptive Load Balancing for MPI Programs. In *Proceedings of the International Conference on Computational Science, San Francisco, CA, LNCS 2074*, pages 108–117, May 2001.

[13] Milind A. Bhandarkar. *Charisma: A Component Architecture for Parallel Programming*. PhD thesis, Dept. of Computer Science, University of Illinois, 2002.

[14] William Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, Bill Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. Polaris: Improving the effectiveness of parallelizing compilers. In *Proceedings of 7th International Workshop on Languages and Compilers for Parallel Computing*, number 892 in Lecture Notes in Computer Science, pages 141–154, Ithaca, NY, USA, August 1994. Springer-Verlag.

[15] Hans Boehm. Advantages and disadvantages of conservative garbage collection. URL: http://www.hpl.hp.com/personal/Hans_Boehm/gc/issues.html.

[16] Hans Boehm. Overview of Boehm-Demers-Weiser conservative garbage collector. URL: http://www.hpl.hp.com/personal/Hans_Boehm/gc/.

[17] Robert K. Brunner and Laxmikant V. Kalé. Handling application-induced load imbalance using parallel objects. In *Parallel and Distributed Computing for Symbolic and Irregular Applications*, pages 167–181. World Scientific Publishing, 2000.

[18] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Techniques for reducing consistency-related communications in distributed shared memory systems. *ACM Transactions on Computers*, 13(3):205–243, Aug. 1995.

[19] Bradford L. Chamberlain, Sung-Eun Choi, Steven J. Deitz, and Lawrence Snyder. The high-level parallel language ZPL improves productivity and performance. In *Proceedings of the IEEE International Workshop on Productivity and Performance in High-End Computing*, 2004.

[20] Ariel Cohen. RDMA offers low overhead, high speed. *Network World*, March 2003. URL http://www.nwfusion.com/news/tech/2003/0324tech.html.

[21] Steven J. Deitz. Tired of MPI: The pocket guide to ZPL, 2003.

[22] Jayant DeSouza and L. V. Kalé. Jade: A parallel message-driven Java. In *Proc. Workshop on Java in Computational Science, held in conjunction with the International Conference on Computational Science (ICCS 2003)*, Melbourne, Australia and Saint Petersburg, Russian Federation, June 2003.

[23] Jayant DeSouza and Laxmikant V. Kalé. MSA: Multiphase specifically shared arrays. In *Proceedings of the 17th International Workshop on Languages and Compilers for Parallel Computing*, West Lafayette, Indiana, USA, September 2004.

[24] W. Fenton, B. Ramkumar, V.A. Saletore, A.B. Sinha, and L.V. Kale. Supporting machine independent programming on diverse parallel architectures. In *Proceedings of the International Conference on Parallel Processing*, pages 193–201, St. Charles, IL, August 1991.

[25] F.H.McMahon. The livermore fortran kernels: A computer test of the numerical performance range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, Livermore, California, December 1986.

[26] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[27] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler support for machine independent parallel programming in fortran-d. In J. Salz and P. Mehrotra, editors, *Compilers and Runtime Software for Scalable Multiprocessors*. Elsevier Science Publishers B.V., 1992.

[28] Chao Huang, Orion Lawlor, and L. V. Kalé. Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03)*, College Station, Texas, October 2003.

[29] Galen C. Hunt and Michael L. Scott. The coign automatic distributed partitioning system. In *Third Symposium on Operating Systems Design and Implementation*, pages 187–200, New Orleans, Louisiana, USA, February 1999.

[30] Y.-S. Hwang, R. Das, J.H. Saltz, M. Hodoscek, and B.R. Brooks. Parallelizing Molecular Dynamics Programs for Distributed Memory Machines. *IEEE Computational Science & Engineering*, 2(2):18–29, Summer 1995.

[31] L. Iftode and J. P. Singh. Shared virtual memory: Progress and challenges. *Proc. of the IEEE, Special Issue on Distributed Shared Memory*, 87(3):498–507, 1999.

[32] Rashmi Jyothi, Orion Sky Lawlor, and L. V. Kale. Debugging support for Charm++. In *PADTAD Workshop for IPDPS 2004*, page 294. IEEE Press, 2004.

[33] L. V. Kale, Milind Bhandarkar, and Robert Brunner. Run-time Support for Adaptive Load Balancing. In J. Rolim, editor, *Lecture Notes in Computer Science, Proceedings of 4th Workshop on Runtime Systems for Parallel Programming (RTSPP) Cancun - Mexico*, volume 1800, pages 1152–1159, March 2000.

[34] L. V. Kale, Milind Bhandarkar, Narain Jagathesan, Sanjeev Krishnan, and Joshua Yelon. Converse: An Interoperable Framework for Parallel Programming. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 212–217, April 1996.

[35] L. V. Kalé, Milind Bhandarkar, and Terry Wilmarth. Design and implementation of parallel java with a global object space. In *Proc. Conf. on Parallel and Distributed Processing Technology and Applications*, pages 235–244, Las Vegas, Nevada, July 1997.

[36] L. V. Kale and Sanjeev Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

[37] Laxmikant Kalé, Robert Skeel, Milind Bhandarkar, Robert Brunner, Attila Gursoy, Neal Krawetz, James Phillips, Aritomo Shinozaki, Krishnan Varadarajan, and Klaus Schulten. NAMD2: Greater scalability for parallel molecular dynamics. *Journal of Computational Physics*, 151:283–312, 1999.

[38] Laxmikant V. Kalé, Sameer Kumar, and Jayant DeSouza. A malleable-job system for timeshared parallel machines. In *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2002)*, May 2002.

[39] L.V. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[40] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proc. of the Winter 1994 USENIX Conference*, pages 115–131, 1994.

[41] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Trans. on Parallel and Distributed systems*, 2(4):440–451, 1991.

[42] C.H. Koelbel, D.B. Loveman, R.S. Schreiber, G.L. Steele Jr., and M.E. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1994.

[43] J. Maassen, R. van Nieuwpoort, R. Veldema, H. Bal, and A. Plaat. An efficient implementation of Java's remote method invocation. In *Proc. Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, pages 173–182, Atlanta, GA, May 1999.

[44] Jason Maassen, Rob van Nieuwpoort, Ronald Veldema, Henri Bal, Thilo Kielmann, Ceriel Jacobs, and Rutger Hofman. Efficient Java RMI for parallel programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(6):747–775, November 2001.

[45] M. W. Macbeth, K. A. McGuigan, , and P. J. Hatcher. Executing Java threads in parallel in a distributed-memory environment. In *Proc. CASCON 98*, pages 40–54, Missisauga, ON, Canada, 1998.

[46] Seung-Jai Min and Rudolf Eigenmann. Combined compile-time and runtime-driven, pro-active data movement in software DSM systems. In *LCR 2004: Seventh Workshop on Languages, Compilers, and Run-time Support for Scalable Systems*, Houston,TX, Oct 2004.

[47] John Mitchell, Terence Parr, et al. Java grammar for ANTLR. URL: http://www.antlr.org/grammars/java.

[48] J. E. Moreira, S. P. Midkiff, M. Gupta, P. V. Artigas, M. Snir, and R. D. Lawrence. Java programming for high-performance numerical computing. *IBM Systems Journal*, 39(1):21–56, 2000.

[49] Christian Nester, Michael Philippsen, and Bernhard Haumacher. A more efficient RMI for Java. In *Proc. ACM 1999 Java Grande Conference*, San Francisco, CA, June 1999.

[50] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global arrays: A non-uniform-memory-access programming model for high-performance computers. In *Journal of Supercomputing*, volume 10, pages 169–189, 1996.

[51] Numrich and Reid. Co-Array Fortran for parallel programming. *ACM Fortran Forum*, 17(2):1–31, 1998.

[52] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, Version 1.2 edition, December 1993.

[53] Parallel Programming Laboratory, University of Illinois, Urbana, IL. *The Charm++ Programming Language Manual*, version 5.8 (release 1) edition, July 2004.

[54] Terence Parr et al. Website for the ANTLR translator generator. URL: http://www.antlr.org/.

[55] Michael Philippsen and Bernhard Haumacher. More efficient object serialization. In *Parallel and Distributed Processing, LNCS, International Workshop on Java for Parallel and Distributed Computing*, volume 1586, pages 718–732, San Juan, Puerto Rico, April 1999.

[56] Michael Philippsen and Matthias Zenger. JavaParty – transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1125–1242, November 1997.

[57] James Phillips, Gengbin Zheng, and Laxmikant V. Kalé. Namd: Biomolecular simulation on thousands of processors. In *Workshop: Scaling to New Heights*, Pittsburgh, PA, May 2002.

[58] James C. Phillips, Gengbin Zheng, Sameer Kumar, and Laxmikant V. Kalé. NAMD: Biomolecular simulation on thousands of processors. In *Proceedings of SC 2002*, Baltimore, MD, September 2002.

[59] S. J. Plimpton and B. A. Hendrickson. A new parallel method for molecular-dynamics simulation of macromolecular systems. *J Comp Chem*, 17:326–337, 1996.

[60] J. Ramanujam. Integer lattice based methods for local address generation for block-cyclic distributions. In D. Agrawal and S. Pande, editors, *Languages, Compilation Techniques and Run Time Systems for Scalable Parallel Systems*. Springer-Verlag, 1998.

[61] Joel Saltz, Kathleen Crowley, Ravi Mirchandaney, and Harry Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8:303–312, 1990.

[62] A. Sinha and L.V. Kalé. Information Sharing Mechanisms in Parallel Programs. In H.J. Siegel, editor, *Proceedings of the 8th International Parallel Processing Symposium*, pages 461–468, Cancun, Mexico, April 1994.

[63] John Skeet and Dale King. Java parameter passing. URL: http://www.yoda.arachsys.com/java/passing.html#formal.

[64] Jimmy Su and Katherine Yelick. Array prefetching for irregular array accesses in Titanium. In *Sixth Annual Workshop on Java for Parallel and Distributed Computing*, Santa Fe, NM, April 2004.

[65] Sun Microsystems, Inc., Mountain View, Calif. *Remote Procedure Calls: Protocol Specification*, May 1988.

[66] E. Tilevich and Y. Smaragdakis. J-orchestra: Automatic java application partitioning, 2002.

[67] L. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8), August 1990.

[68] Wikipedia. Overview of garbage collection. URL: http://en.wikipedia.org/wiki/Computer_memory_garbage_collection.

[69] K. A. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. N. Hilfinger, S. L. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect. *Concurrency: Practice and Experience*, 10(11–13), September – November 1998.

[70] Weimin Yu and Alan Cox. Java/DSM: A platform for heterogeneous computing. In *ACM 1997 Workshop on Java for Science and Engineering Computation June 21-97 PPoPP Las Vegas*, Las Vegas, June 1997.

[71] Gengbin Zheng, Lixia Shi, and Laxmikant V. Kalé. Ftc-charm++: An in-memory checkpoint-based fault tolerant runtime for charm++ and mpi. In *Cluster 2004*, number 04-06, San Francisco, CA, September 2004.

Author's Biography

Jayant DeSouza spent the first 21 years of his life in the city of Madras, India (now known as Chennai). There, he completed his schooling at Don Bosco school (Egmore), and obtained a B. E. in Computer Science Engineering from the University of Madras at Sri Venkateswara College of Engineering in 1993.

He then came to the USA where he has lived for the last eleven years. In 1993 he started his Master of Science degree in Computer Science at Kansas State University, Manhattan("the little apple"), Kansas, which he completed in 1997, and in 1994 began working full-time at AT&T Bell Laboratories, Naperville, IL.

In 1995, he joined the University of Illinois at Urbana-Champaign to begin his PhD in Computer Science. From 1998–2000 he took a leave of absence from the University and worked as a Senior Software Engineer at Guidant Corp., a medical devices company, in St. Paul, MN.

At the University of Illinois, Jay has been with the Parallel Programming Laboratory (PPL) led by Prof. L. V. Kalé since Fall 1996. He worked on the original Charm++ translator, and then on the design of an improved translator that used a proxy mechanism for parameter marshalling and other features. This interest in programming languages for improving productivity in parallel computing let to his doctoral work on the Jade language and the MSA specifically shared memory programming paradigm. At PPL, he was also one of the primary developers of the Faucets anonymous grid computing project.

He has been a Research Assistant at the Theoretical and Computational Biophysics Group (TCBG) at the Beckman Institute for Advanced Science and Technology since 2000,

where he worked on the BioCoRE collaborative environment project and on small molecule systems performance studies for the NAMD molecular dynamics application.

Over the years, he has completed internships at Silicon Graphics Inc., Mountain View, CA, Cadence Design Systems, San Jose, CA, and KAI Software Lab, Intel Corp., Champaign, IL. After completion of his PhD, he will be joining KAI Software Labs, Intel Americas, Inc., in Champaign, IL, as Senior Software Engineer.